X68k

Programming Series

吉野智興＋中村祐一＋石丸敏弘＋今野幸義……共著

(#1)

X680x0 Develop.

Vol. 1

Programmer's Guide

Manual Books

SOFT BANK

X68k

Programming Series

吉野智興＋中村祐一＋石丸敏弘＋今野幸義……共著

(#1)

X68000 Develop.

Vol. 1

Programmer's Guide

本書は,「*X680x0 Develop. & libc II*」の内容に即して「*X68000 Develop.*」を加筆修正したものです。
便宜上，文章中「X680x0 ***では」と記載されているのは,「*X680x0 Develop. & libc II*」に収録されている各ツールを指し,「従来では」または「X68000 *」と記載されているものは,「*X68000 Develop.*」に収録されている各ツールを指します。**

- 本書に記載したすべてのプログラムは,「GNU 一般公有使用許諾書」を適用しています。
- その他のシステム名，CPU 名などは一般に各社の登録商標です。本文中では，とくに TM, Ⓡ は明記しておりません。

はじめに

『X68k Programming Series』は，サードパーティーからの開発環境の提供がまったくない **X68000** シリーズのプログラミング環境改善を目指して企画された書籍ソフトウェアです。最初の企画では，書籍ではなく *Oh!* X の別冊として進めていたのですが，ソフトウェアの規模が大きく，ドキュメントの量も別冊といった形では多すぎるため，書籍として皆様にお届けするようになりました。本書と付属のソフトウェアはその企画の第 1 弾になります。

基本開発ツールとしてコンパイラ，アセンブラ，リンカ，デバッガを集め，各制作者がそれぞれ解説を執筆するという，ほぼ理想的な形で作り上げることができました。純正品では明確にされていない内部的な仕様等も，本書では文書化されています。これらのソフトウェアが **X68000** のプログラミング環境の向上に少しでも役に立つことを期待しています。

1993 年 1 月吉日

著者を代表して

Manual Books 発刊へのまえがき

「**SHARP** 純正開発環境をぶっとばせ」(!?) で作成された「*X680000 Develop.*」も **X68030** の登場とともにバージョンアップしました。ところがこのバージョンアップ版である「*X680x0 Develop. & libc II*」はあくまでも前著の追補版であって，今まで使っておられた人はともかく，追補版のみ入手した人には「まともなマニュアルがない」といったことが起こるのです。そこで，前著「*X68000 Develop.*」からディスクを除いて提供しようということになったのが，マニュアル版「*X680x0 Develop. Manual Books*」です。バージョンアップは，前著「*X68000 Develop.*」を基本的に踏襲して行われていますので，細かい違いのみマニュアル版では補足するにとどめてあります。ですから，本書で補足されている機能説明は，全部「*X680x0 Develop. & libc II*」に記載されています。

おかげさまで，X68k Programming Series は開発環境として，ほぼ「標準的」な地位を獲得することができました。ですが，バグが完全に消滅したわけでもないでしょうし，私たちはそれをフィックスする義務を負った立場でもありません。このような無保証なソフトウェア群が純正の開発環境を凌駕するまでに成長したのは，多くの **X680x0** でプログラムすることをこよなく愛する人々の応援があったからです。「*X68000 Develop.*」を発刊した後も，これらの方々からたくさんのバグレポートや改良改善の提案がありました。できるだけその要望にお答えできるようにしたつもりですが，まだ不備な点があるかも知れません。そういった場合には，あくまで無保証のフリーソフトウェアである点を思い出して対処されることを望みます。

残念なことに，最近では **X680x0** 向けの市販のソフトウェアが少なくなっているのは事実です。ですが，**X680x0** シリーズは趣味でソフトウェアやハードウェア改造 (!?) を行うには非常におもしろい機械です。そのような土壌のなかで育った開発環境ですので，たぶんプログラムを楽しむには十分に満足してもらえると信じています。

1994 年 8 月吉日

著者を代表して

X68k Programming Series #1

X680x0 Develop.

Vol.1 Programmer's Guide.

C O N T E N T S

Chapter 1

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Chapter 6

Chapter 7

Chapter 1

X68000開発ツール概説

本章では，初めてＣおよびアセンブラなどの言語を使用するユーザのために，基本的な事項について説明します。

1.1 ……………………………………… インストール

まず，付録ディスクのバックアップを作成して，元ディスクは大切に保管しておいてください。当然ですがコピープロテクトはかかっていませんので，普通にバックアップできます。インストール方法は，付属の A ディスクに入っている inst.x を起動するだけです。inst.x が起動すると，プログラムが次々と質問してきますので，自分の環境に応じて選択してください[1)]。その際，立ち上げディスクの CONFIG.SYS と AUTOEXEC.BAT を書き換える場合があります。ユーザの中には，これらのファイルを勝手に書き換えられると，何らかの支障をきたす場合もあると思います。そのようなときは，あらかじめ CONFIG.SYS と AUTOEXEC.BAT のバックアップを取っておくか，またはオートインストールせずに独自にインストールしてください。なおインストーラを使わない場合は，A ディスクに入っている README をよく読んで，適切にインストールするようにしてください[2)]。

1)インストールする前に，必ず自分のマシンのハード環境と立ち上げディスクの設定環境を把握しておいてください。

2)オートインストールする場合も，A ディスクの README はよく読んでください。インストールに関する情報は，すべてこのファイルで提供します。

1.2 ……………………………………………… 環　境

“環境” という言葉をソフトウェアで使う場合には，いろいろな意味があります。**X68000** という機械を使うのも環境ですし，**X68000** を使う場合に「COMMAND.X を常用する」,「VS.X を常用する」あるいは「SX–Window を常用する」などもすべて環境と呼ばれます。大別すれば，

- **COMMAND.X を使うコマンドライン主体の環境**
- **SX–Window，VS.X などのビジュアル的な環境**

の 2 つに分かれます。そしてこの開発ツールキットは，前者のコマンドライン主体の開発環境として提供されています。このようなコマンドライン主体の開発環境はどちらかといえば初心者向きではなく，ある程度の経験と知識が必要になります。開発環境として特に著名なものに，エンジニアワークステーション上の **UNIX** がありますが，普通のユーザには簡単にはなじめないマニア向け[1]の環境です。初めて **X68000** でプログラムをする場合には，この “環境” についてある程度の知識が必要となります。

1)cat, ls, grep 等，名前だけでは何をするのかを推測すらできないコマンドが乱舞します。

1.2.1　OS 環境

コンピュータは OS と呼ばれる基本ソフトウェアがなければ，ただの金属と合成樹脂の物体でしかありません。**X68000** にフォーマットされていないフロッピーディスクを挿入して電源を入れると，**X68000** は「正しいディスクを入れてください」とだけ表示して，停止してしまいます。“正しいディスク” というのは，この基本的なソフトウェアが記録されたフロッピーディスクのことです。

ここで **X68000** の標準 OS である **Human68k** について，電源を投入してから動き出すまでの流れを見てみましょう。

1. **電源投入**
   電源投入で **X68000** の頭脳 68000 が動き出す
2. **ハードウェア初期化**
   ディスプレイにゴミが表示されたりしないように，さまざまな初期化作業を行う

3. 増設ハードウェアの確認

電源が入っていなくても消えない不揮発メモリから，ハードディスクの有無や内蔵メモリの量を確認する

4. ブートデバイスの決定

同じ不揮発メモリから，どこから最初に実行するプログラムを読むのかを決める[2)]

2)読み出す補助記憶装置をブートデバイスと呼びます。

ここまでは，ハードディスクの電源が入っていなかったり，フロッピーディスクが挿入されていない状態でも必ず行われます。そしてこの段階まで処理するプログラムは，内蔵されている ROM[3)]に書かれています。このように最初から ROM として備わっているプログラムをファームウェアと呼びます。昔の 8 ビットパソコンや今の某国民機種では，このファームウェアとして BASIC が組み込まれていますので，電源を入れれば BASIC がすぐに使えます。しかし，**X68000** では本体だけではここまでしか動きません[4)]。**X68000** では，IOCS コールはファームウェアです。つまり，これは特に **Human68k** が存在しなくても使えるシステムサービスです。市販品で「**Human68k** で読めないディスクなのに，ちゃんとディスクから立ち上がってゲームができる」ソフトウェアがありますが，これはこの IOCS コールをおもに使って，**Human68k** に代わる独自の OS を用意しているからです。

3)Read Only Memory の略。

4)マニアなら設定しだいで，ROM ディスクからブートできることはご承知でしょう。

さてここからが，本当の立ち上げ作業になります。ブートデバイスの用意が確認できたら，ファームウェアはそのデバイスからイニシャルプログラムローダと呼ばれるプログラムを読み出します。このプログラムは各デバイスごとに所定の位置が決まっており，そこから読み出しを行い，もし正常に読み出せたら実行します。このプログラムのことをしばしば“IPL”と呼びます。**Human68k** では，この IPL は **Human68k** の本体である HUMAN.SYS をメモリに読み出すだけのプログラムです。したがって，正常に読み出せたら IPL を実行します。HUMAN.SYS は，おおよそ次のような作業を行います。

1. DOSCALL のサービス登録

Human68k が行う処理を割り込み処理として登録する[5)]

5)具体的に説明すると，“未定義命令 $FF 処理ベクタの登録”を行います。

2. CONFIG.SYS の処理

ブートデバイスから CONFIG.SYS という名前のファイルを捜して，存在すればその記述内容に従って処理する

3. SHELL の起動

CONFIG.SYS に指定があればそのプログラムを，なければ COMANND.X を起動する

ここまでの作業を行い，初めて **Human68k** は立ち上がりました。さらに起動された COMMAND.X は，AUTOEXEC.BAT というファイルを捜して実行します。ここでようやく，“A:>”という文字が画面に出て，次の命令を待つ状態になります。**X68000** の別の OS である OS9/68000 では，IPL が別のプログラム，つまり OS9/68000 の本体を読み出すようになっていて，まったく別の顔を見せるようになります。これからプログラムを学ぼうとされているユーザにとっては，

「なんて難しい言葉が並んでいるんだろう」と思われたことでしょう。でも実際に **Human68k** が動くまでには，実はもっともっとたくさんのプログラムが実行されていて，もっともっと複雑な過程を経由しているのです。しかし，最終的に COMMAND.X が命令待ちになるまでの過程というのは，DOS マシンでも **UNIX** マシンでもそう大差ありません。

この開発ツールを使ううえで重要なのは，HUMAN.SYS が読む CONFIG.SYS と COMMAND.X が最初に実行する AUTOEXEC.BAT です。この 2 つのファイルが **X68000** の “環境” を決定する重要なファイルなのです。これらのファイルに記述する事柄については，ぜひ **Human68k** のマニュアルを参照して熟知されることをお勧めします。

1.2.2　環境変数

環境変数というのは，あるソフトウェアが動く場合にそのソフトがどのような “環境” で動くのかを知るためのシステム変数です。たとえば，あるスクリーンエディット可能なソフトウェアを考えてみます。某国民機種のごとく画面が横 80 文字，縦 25 文字と割り切って作成されたソフトウェアでは，**UNIX** のようにさまざまな大きさの画面をもった別の種類のマシンでも同じように動作させることは不可能です。**UNIX** ではこのような場合，画面の大きさやカーソル移動の方法などを記述したファイルを用意しておき，その名前をある環境変数に登録しておきます。ソフトウェアはその環境変数を見て，画面情報を取得し，適切なカーソル移動や画面表示を行います。環境変数は，このような各個人のもつさまざまな趣味やハードの性質を，ソフトウェア側で吸収するための情報を登録しておくために用いられます。

Human68k の COMMAND.X では，この環境変数を “SET” コマンドで登録や削除ができます。

```
A>SET MARIKO=A
A>SET
MARIKO=A
A>SET MARIKO=
A>SET
A>
```

普通このような環境変数は，AUTOEXEC.BAT で登録するか，CONFIG.SYS で環境ファイルを指定することで登録しておきます。具体的な記述方法については，**Human68k** のマニュアルを参照してください。

1.2.3 環境変数 PATH

環境変数の path (または PATH) は，Human68k が実行可能なファイルを検索する場合に用いる環境変数です。あるソフトをインストールして実行しようとしたとき「コマンドまたはファイル名が違います」と怒られるケースというのは，この環境変数の設定がまちがっている場合がほとんどです。つまりインストールされたディレクトリかドライブが，環境変数に登録されていないということなのです。また，環境変数に新しいディレクトリを登録することを“パスを通す”と表現します。“XXX が動かないんだけどぉ?”“そこにパス通ってるぅ?”という会話はここからきています。さらに X68000 では，“path”と“PATH”が別扱いになりますので注意してください。COMMAND.X では特に問題がなかったように記憶していますが，パソコン通信などで配布されているさまざまなシェルを使う際には注意してください。

1.3 …………………………………………… C 言語

C 言語はアセンブラと Pascal などの“高級言語”との中間にある言語です。**UNIX** の大部分がこの C 言語で記述されたこともあり，この言語が動かないコンピュータを捜すのが難しいくらい普及している言語です。ですが C 言語は，普及しているわりには“難しい言語”とされています。元来がシステム記述用に設計されているので，まちがったプログラムに対しての安全性が極端に省略されていて，非常に簡単に暴走するのが，“難しい”とされる原因の 1 つでしょう。またポインタの概念が，アセンブラを経験していない場合には，わかりにくい点も“難しさ”の要因と考えられます。このセクションでは，**X68000** での C 言語の位置について考えてみましょう。

1.3.1　C 言語のメリット

X68000 で最初にプログラムをする場合に使われているのは，おそらく X–BASIC でしょう[1)]。X–BASIC は他の BASIC とは極端に異なる BASIC で，見ためは C に非常によく似ています。しかし，X–BASIC はインタプリタと呼ばれる処理系で，XC やこの開発ツールの **GCC** はコンパイラという処理系です。さて，この 2 つの処理系の違いはどこにあるのでしょうか？　下の表を見てください。このように，

まったく正反対の性格をもつ処理系である

と，極端にいえば解釈できます。

1)初代機種からのユーザはアセンブラですね。

Table 1-1● コンパイラとインタプリタ

	インタプリタ	コンパイラ
プログラム	1 処理ずつ解釈しながら実行	一度に解釈して機械語にする
修正について	修正が楽で，修正から実行までの期間が短い	修正すると実行までの時間が長い
実行速度	実行は低速，またプログラムを COMMAND.X から呼び出す形式にはできない	実行速度は速く，COMMAND.X から呼び出せる

X68000 ではアセンブラと C 言語とは親戚みたいな関係であり，特に **GCC** のように極端に最適化が進んだコンパイラの場合，C のソースの行がそのままアセンブラに対応してしまうほどです。そのため，究極の高速化を追及する場合以外には，C 言語がもつ移植性の高さがメリットになるでしょう。暴走に対する安全性については，C 言語とアセンブラとではそう大差はないので“高級言語”の安全性というメリットはほとんどありません。しかし記述性能では，やはりアセンブラに比べれば C 言語のほうがメリットが大きいと考えられるので，**X68000** においては，

- **速度が最重要ならアセンブラ**
- **複雑な処理は C 言語**

という切り分け方で記述するとよいでしょう。また別機種からの移植の際には，C 言語処理系がある場合とない場合とでは雲泥の差があります。

1.3.2 C 言語の概略

この開発ツールキットの C コンパイラ **GCC** は，ANSI 規定に準拠した **ANSI C** になっています。XC は厳密な **ANSI C** ではないので，XC ではエラー報告がないソースでも，**GCC** ではエラーになることがあります。基本的な仕様は『プログラミング言語 C 第 2 版』(共立出版) の仕様に準拠しています。詳しい仕様はそちらを参照していただくことにして，ここではごく基本的な概略と本書との仕様の違いの紹介にとどめます。**GCC** の拡張については，第 2 章「**X68000 GCC**」を参照してください。

◆ ソース文字セット

ソースに使える文字セットは，いわゆる ASCII 文字セットとシフト JIS 漢字コードセットです[2]。通常，漢字コードは文字列リテラルでのみ使えますが，環境変数の設定で識別子文字にも漢字を使うことができます。この開発ツールを使うかぎりにおいては，漢字コードの識別子使用には制限はありません。ただし，**SHARP** 製の製品を混在させる場合は動作の保証がありません。

2)マイクロソフト漢字コードセットともいいます。

❖ ソースに使える文字セット

- ABCDEFGHIJKLMNOPQRSTUVWXYZ
- abcdefghijklmnopqrstuvwxyz
- 0 1 2 3 4 5 6 7 8 9
- _ ! " # $ % & ' ( ) = ~ | ‘ ; : ?
- \ { } [ ] / * - + = < > @
- **その他 X68000 でキーボードから入力できるすべての表示可能文字**

❖ 識別子として使える文字

- ABCDEFGHIJKLMNOPQRSTUVWXYZ
- abcdefghijklmnopqrstuvwxyz

- 0 1 2 3 4 5 6 7 8 9　(ただし識別子先頭は不可)
- **X68000で使えるすべての漢字コード**　(ただし環境変数の設定が必要)
- _　(ただしこの文字で始まる識別子はシステム予約である)

◆ 基本データタイプ

ANSI Cのすべてのデータタイプに加えて“long long int”，“64ビット整数”がサポートされています。ただしXCのライブラリを使う場合は，このデータをprintf()関数などの関数を使って表示させることはできません[3]。

3)パソコン通信などでは，このデータを表示入力させるライブラリが流通しています。入手可能なユーザでしたら，使うことができます。

❖ 8ビット整数

- char　(符号の有無はオプションで指定可能)
- signed char
- unsigned char

❖ 16ビット整数

- short
- short int
- signed short
- signed short int
- unsigned short
- unsigned short int
- int　(オプションによって使用可能)

❖ 32ビット整数

- int
- long
- long int
- signed int
- signed long
- signed long int
- unsigned int
- unsigned long
- unsigned long int

❖ 64ビット整数

- long long
- long long int
- signed long long
- signed long long int
- unsigned long long
- unsigned long long int

❖ 32ビット浮動小数点

- float

❖ 64 ビット浮動小数点

- double
- long double　(GCC では long double は double と等価)

◆ 予約語

すべての **ANSI C** の予約語だけではなく，GNU 拡張による予約語と **X68000** 専用拡張のための予約語が増加しています。拡張予約語は，オプションを指定すれば，**ANSI C** で規定された "_" で始まる形式の予約語だけ認識するようにできます。また，XC のように非 **ANSI C** なシステムでもコンパイル可能なように，一部の **ANSI C** 予約語が "_" を付加した予約語にも登録されています。

❖ GCC の予約語種別

- __alignof　**GCC 拡張予約語**
- __alignof__　**GCC 拡張予約語**
- __asm　**GCC 拡張予約語**
- __asm__　**GCC 拡張予約語**
- __attribute　**GCC 拡張予約語**
- __attribute__　**GCC 拡張予約語**
- __common　**X68000 GCC 拡張予約語**
- __common__　**X68000 GCC 拡張予約語**
- __const　**ANSI C 代替予約語**
- __const__　**ANSI C 代替予約語**
- __inline　**GCC 拡張予約語**
- __inline__　**GCC 拡張予約語**
- __relocate　**X68000 GCC 拡張予約語**
- __relocate__　**X68000 GCC 拡張予約語**
- __remote　**X68000 GCC 拡張予約語**
- __remote__　**X68000 GCC 拡張予約語**
- __signed　**ANSI C 代替予約語**
- __signed__　**ANSI C 代替予約語**
- __typeof　**GCC 拡張予約語**
- __typeof__　**GCC 拡張予約語**
- __volatile　**ANSI C 代替予約語**
- __volatile__　**ANSI C 代替予約語**
- __DOSCALL　**X68000 GCC 拡張予約語**
- __DOSCALL__　**X68000 GCC 拡張予約語**
- __SXCALL　**X68000 GCC 拡張予約語**
- __SXCALL__　**X68000 GCC 拡張予約語**
- asm　**GCC 拡張予約語**
- auto
- break

- case
- char
- common
- const
- continue
- default
- do
- double
- else
- enum
- extern
- float
- for
- goto
- if
- inline
- int
- long
- register
- relocate
- remote
- return
- short
- signed
- sizeof
- static
- struct
- switch
- typedef
- typeof
- union
- unsigned
- void
- volatile
- while
- DOSCALL　　　X68000 GCC 拡張予約語
- SXCALL　　　X68000 GCC 拡張予約語

◆ 定数表現

X68000 GCC と XC とでは，定数表現の扱いが全然違います。**GCC** は **ANSI C** なので，すべての整数定数表現は "`unsigned int`" として扱われます。このことは，整数定数を `signed int` の表現し得る限界点で顕著な違いとして現れます。たとえば `-2147483648` は，通常の 2 の補数表現 32 ビット整数の最小値です。XC でこの数値を処理する場合，そのまま `-2147483648` として扱いますが，**GCC** では `2147483648` に単項演算子 "`-`" を施したものとして認識されます。`2147483648` は `int` では表せませんので，`unsigned int` になります。この点に十分に注意してください。また XC の `limits.h` は，**GCC** では正しい整数の範囲を示せないことにも注意してください。

◆ 記憶クラス

GCC では，XC で未サポートの記憶クラスもサポートしています。また **X68000 GCC** の拡張である SX–Window モードでは，いくつかの記憶クラスが追加されています[4]。また **GCC** で最適化を指定した場合，`register` および `auto` 記憶クラスは，ほぼ完全に無視されてしまいます。つまりコンパイラは，自分の判断でレジスタに変数を割り当てるのです。ただし **GCC** は，レジスタ割り当て情報をデバッグコードとして出力しますので，ソースコードデバッグには支障はありません。

4) これらについての詳しい内容については，第 2 章「X68000 GCC」を参照してください。

1.3.3 C 言語とアセンブラ

C 言語を学ぶ場合には，いくつかのハードルがあります。すべてのハードルを乗り超えるお手伝いはこのマニュアルではできませんので，ここでは本セクションの冒頭で指摘した C 言語のポインタについてだけ，アセンブラとの対比で説明します。

◆ 基礎知識 (1)

どんなコンピュータにも "メモリ" があります。メモリというのは，数値を記録しておく箱と考えてください。箱の 1 番小さい単位は 8 ビットの数値を記憶でき，これを 1 バイトと呼びます。カタログに載っているメインメモリ 2M バイトというのは，この 8 ビットの箱が 2,097,151 個あることになります。この箱には全部番号がついていて，これが "アドレス" と呼ばれる数値です。

CPU には "レジスタ" といって，このメモリと数値を入れたり，出したり，加工したりする場所があります。**X68000** の CPU 68000 では，このレジスタが "16 個" あり，d0 から d7，a0 から a7 という名前がついています。さらに，CPU によって「レジスタで同時にいくつまでのメモリが扱える」かが決まっていて，68000 では同時に "4 つ" まで扱うことができます。メモリは最小単位が 8 ビットで 1 箱ですから，1 個，2 個，4 個 をそれぞれ集めた単位に名前がつけられています[5]。

5) 3 個がない点で悩まないでくださいね。

- **メモリ 1 箱 ：　8 ビット ： 1 バイト**
- **メモリ 2 箱 ： 16 ビット ： 1 ワード**
- **メモリ 4 箱 ： 32 ビット ： 1 ロングワード**

これらは **X68000** では「なんと C の整数数値の種類と一致」しています。それぞれ char, short, int です。これは偶然でもなんでもなく，C 言語において int は，その CPU が扱うのに最も適したビット幅をもつことになっているからです。つまり 68000 では，レジスタの幅は“32 ビット 1 ロングワード”になっています。その意味では，68000 は 32 ビットマシンなのですが，モトローラは謙虚にもハードウェアで同時に扱える幅を採用して，68000 は 16 ビット CPU に分類しています。もう少しくだいて説明すると「同時に 4 つ扱えるけれども，メモリとレジスタで値を出したり，入れたりする場合には，2 個ずつ 2 回に分けて行っている」ということなのです。ちなみに某国民機種では，32 ビットというロゴがついたマシンでも，メモリ 2 個 16 ビットしか同時に扱えない状態でたいていは動いています。

◆ 基礎知識 (2)

メモリには番号がついていて，それをアドレスと呼ぶことは「基礎知識 (1)」で説明しました。番号は数値ですからレジスタで扱えます。つまりレジスタで箱の番号を指定して，数値を出したり，入れたり，加工したりすることができるのです。68000 では，レジスタは 32 ビットですからアドレスも 32 ビットで扱えるのですが，実際のハードウェアでは上位の 8 ビットが出力されていないので，メモリの箱の数は 24 ビットに制限されています。このことはレジスタで箱の番号，つまりアドレスを指定する場合には，上位の 8 ビットが無視されるということになります。これらの制約のなかで，68000 ではメモリの箱の番号を指定して値を取り出す方法がたくさんあります。この方法をアドレス形式という頭の痛くなる言葉で表現します。アドレス形式の詳細については Appendix[6] で詳しく説明してありますので，ここでは省略します。

6)P.287 参照。

もう 1 つの基礎知識として，レジスタは C 言語においては変数であるという点です。数がかぎられたレジスタを，C ソースに現れてくる変数にいかに効率よく割り当てるかは，論文になるくらいコンパイラにとっては重要な要素の 1 つです。

◆ ポインタとは？

ここまで理解できれば，ポインタは全然怖くありません。C 言語においてポインタとは，「基礎知識 (1)」で説明したアドレスそのものを扱う変数のことです。たとえば int *x というのは，1 ロングワードの整数のアドレスを入れておく変数になります。アセンブラで，a0 で指定されたアドレスから 1 ロングワードを取り出して，d0 に格納する作業は “move.l (a0),d0” です。この作業は，C 言語において List 1-1 とまったく等価になります。

List 1-1● int ポインタ

```
1:  /* メモリ x 番地 を取り出して返す */
2:  int val (int *x)
3:  {
4:    return *x;
5:  }
```

では，ポインタ変数に対する演算とは何でしょうか？　答は簡単ですね。出したり，入れたり，加工したりするメモリの加工対象アドレスを変更するという操作のことなのです。ポインタはその名前のとおり，あるメモリを指し示す変数です。x がアドレスで，*x はそのアドレスにあるメモリの内容になります。ここで問題になるのが「そのポインタが示すメモリにはどんなデータが入っている」かなのです。List 1-1 では，変数 x は int を指し示す変数なので，*x は メモリ 4 箱分の 1 ロングワードの値をとることになります。では“x + 1”は何でしょうか？　答は x の値[7]の 4 バイト先のアドレスになります。なぜ 4 バイト先になるのかというと，x は int のアドレスを示す変数ですから，その 1 つ先は int のサイズ 4 バイト分だけ進んだアドレスだからです。

7)*x ではないことに注意。

また C 言語で最も難解なのが“->”演算子でしょう。これは構造体のポインタで頻繁に用いられます。しかしこれは「ポインタがアドレスそのものである」と考えれば理解できると思います。

List 1-2● 構造体ポインタ

```
1:  struct point {
2:    int x;
3:    int y;
4:  };
5:  int val_x (struct point *P)
6:  {
7:    return P -> x;
8:  }
```

List 1-2 において，7 行目の“P -> x”の部分を「P が示すアドレスにある x の値」と読めば，簡単に理解できます。変数 P は構造体 point を指し示すアドレスを表す変数で，それが指し示す構造体の変数メンバ x を参照することになります。

“P -> y”はどうなるのでしょうか？　この場合アセンブラで，P の値が a0 に格納されていれば，

```
move.l 4(a0),d0
```

に相当します。構造体のメンバ“int x”のサイズ分だけオフセット[8]したアドレスが，メンバ y のアドレスになるからです。では P++ では，ポインタ変数 P は整数にしていくつ増加するでしょうか？　答は，構造体のサイズ分の 8 増加します。

8)“ずれた”ですね。

最後に，最も C 言語で陥りやすい罠である配列とポインタの違いについて説明します。ここまでで「ポインタはアドレスを示す変数」であることは理解できたと思います。最も頻繁にでてくるまちがいが List 1-3 です。

List 1-3 ● 未初期化ポインタ

```
 1:  #include <stdio.h>
 2:
 3:  main ()
 4:  {
 5:    char *ptr;
 6:    strcpy (ptr, "PC9801");
 7:    printf ("%s\n", ptr);
 8:  }
```

ライブラリマニュアルを見て「ふむふむ **strcpy ()** 関数 は 2 つの **char *** の引数で，1 番目の引数に 2 番目の引数の文字をコピーする，ふむふむ」で作られたのがこのプログラムです。怖いのは，この手のまちがったプログラムがいわゆる初心者向け C 言語入門解説書に堂々とサンプルとして掲載されている場合があることです。**char *ptr** は **char** のアドレスを保持する変数で，その示しているアドレスは List 1-3 では不定です。その意味不定のアドレスに，文字列をコピーしています。ポインタ変数はアドレスを示す変数であって，そのアドレスが正当なアドレスであるという保証は，プログラマが責任をもって記述しなければならないことです。List 1-4 では，**char *ptr** は配列 **buff** のアドレスを示すように初期化されていて，かつ，その大きさは文字列を格納するのに十分な大きさがあるので問題ありません。

List 1-4 ● 初期化済みポインタ

```
 1:  #include <stdio.h>
 2:
 3:  char buff[128];
 4:
 5:  main ()
 6:  {
 7:    char *ptr;
 8:    ptr = buff;
 9:    strcpy (ptr, "PC9801");
10:    printf ("%s\n", ptr);
11:  }
```

それでは「初期化すればいい」ということになり，これまた頻繁に現れるのが List 1-5 です。**char *ptr** は確かに文字列のアドレスで初期化されています。しかし，そのアドレスは文字列リテラルです。**ANSI C** において文字列リテラルを変更しようとする行為は「結果は不定」とされています。

List 1-5 ● 配列とポインタ

```
 1:  #include <stdio.h>
 2:  char *ptr   = "Ｘ６８０００";
 3:  char arry[] = "Ｘ６８０００";
 4:
 5:  main ()
 6:  {
 7:    printf ("%s\n", ptr);
 8:    printf ("%s\n", arry);
 9:    /* ポインタと配列の違いを知らない */
10:    strcpy (ptr, "PC9801");
```

```
11:     /* こちらは問題ない */
12:     strcpy (arry, "PC9801");
13:  }
```

さらに，配列の初期化構文とポインタの初期化構文はそっくりですが，その内容は全然異なります。前述の List 1-5 は，その典型的な勘違いの例です。**X68000** の **GCC** では，まったく同一の文字列リテラルは同じアドレスになるように最適化されます。そのため List 1-5 のような行為をすると，もし同じファイルに同じ文字列があった場合には，そこも同時に書き換えたことになります。MS–DOS の C 言語で作成されたプログラムには，時々このようなポインタ関連のチェックの甘いソースがあります。このようなソースを **X68000** や **UNIX** でコンパイル／実行すると，**X68000** ではバスエラー，**UNIX** ではコアダンプを起こすことがありますので，知っておいて損はないでしょう。

1.3.4 アセンブラによる開発

X68000 の CPU 68000 は簡潔なアーキテクチャで，アセンブラでも比較的容易にプログラミングできます。この開発ツールキットに含まれる **HAS** は強力なマクロ機能を備えていますので，使い方によっては C に匹敵するほど見通しのよい開発も可能でしょう。ただしこの開発ツールキットには，アセンブラのみで作成されたプログラムを効率よくデバッグするデバッガは含まれていません。

アセンブラを学ぶよい方法の 1 つは，**GCC** が生成するアセンブラプログラムを C ソースと対応させて比較することです。もし「ここは，この方法のほうが効率がよい」と思う場合には，それを実際にアセンブラで作成してみるのです。この方法では，**GCC** がどのように関数を呼び出ししているかなどの情報も，同時に学習することができます。

Chapter 2
X68000 GCC

この開発ツールキットの X68000 GCC は，通称 “真里子版 GCC” と呼ばれる GCC です。FSF の GNU CC Ver 1.42 をベースにして X68000 用に大幅に改造したコンパイラですが，今までまとまったドキュメントはありませんでした。本章では， X68000 GCC のすべての機能を文書化しました。

2.1 ……………………………………GCCの概要

GCCはFSFで開発された**UNIX**上のCコンパイラです。それを**X68000**に移植したものが，**X68000 GCC**です。**GCC**とは，CPU 680x0において，おそらく世界最高速のコードを出力するCコンパイラです。

2.1.1 GCCが扱うファイル

GCCコンパイラドライバは，次のファイルを扱うことができます。

❖ **Cソースファイル**
Cプログラムのソースファイルです。拡張子は“.c”です。

❖ **アセンブラソースファイル**
アセンブラのソースファイルです。このソースファイル中に未解決なシンボルがあると，それは外部参照として処理されます。通常，拡張子は“.s”です。**HAS**拡張機能を用いる場合は，拡張子が“.has”になります。

❖ **オブジェクトファイル**
アセンブラソースをアセンブルすることによって得られるファイルです。このファイルはオプションで特に指定がなければ，clibとgnulibの2つのライブラリファイルとともにリンクされます(デフォルト)。拡張子は，“.o”です。

またこの**GCC**コンパイラドライバには，前田氏作成のcshwildライブラリがリンクされています。このライブラリによって，255文字を超えるコマンドラインを，MS-DOSふうにいえば応答ファイルの形で渡すことができます。command.xで渡すことができない255文字を超えるオプションは，そのままそのオプションをファイルにして，次のような書式で渡すことができます。

```
gcc -+-+-<ファイル名>
```

コンパイラドライバは指定されたオプションのほかに，多数の引数をコンパイラに指定することがあります。この場合には，コンパイラドライバが応答ファイルを自動的に作成するので，ユーザが特に意識する必要はありません。しかし，この応答ファイルは環境変数tempが示すディレクトリに作成されるので，ディスクの空き容量には十分注意してください。

2.1.2 GCC が使う環境変数

GCC はコンパイルを行うときにいくつかの環境変数を参照します。環境変数にはコンパイラが稼働するのに必須のものと，そうでないものがあります。必須の環境変数は，すべて XC が必要とするものと同一です。付属ディスクのインストーラを使用した場合には，これらは自動的に適切に設定されます。オートインストールされなかったユーザは，本セクションを参考にご自分の望まれる環境を構築してください。

◆ 必須の環境変数

以下の環境変数は必ず設定しておいてください。 XC が動いている環境ならば，必ず設定されている環境変数です。

❖ **PATH, path**

コンパイラドライバ gcc.x が，プリプロセッサ gcc_cc1.x，コンパイラ本体 gcc_cc1.x を起動する場合に検索するディレクトリです。ドライバ自体も，この環境変数に指定されたディレクトリに存在しているのが普通です。

❖ **include**

システムヘッダ[1]がおかれたディレクトリをフルパスで設定します。

❖ **temp**

コンパイラが作業用テンポラリファイルを作成するディレクトリをフルパスで指定します。できるだけ高速なディスクドライブを指定します。普通は RAM ディスクドライブを設定しておきます。

◆ 必須でない環境変数

ここで説明する環境変数は **X68000** 独自のものです。環境変数名には移植者の“悪趣味”も反映されています[2]。

❖ **GCC_OPTION**

環境変数“GCC_OPTION”によって，あらかじめいくつかのオプションスイッチを設定しておくことができます。いつも同じオプションを設定する場合や，タイピング節約およびコマンドラインの文字数節約[3]に有効です。

```
set GCC_OPTION= AEFGILMPST
```

GCC_OPTION, AEFGILMPST はすべて大文字で指定しなければならず，“=”と“GCC_OPTION”との間にスペースを入れると無効になります。

また -f がついたオプションは，それぞれコマンドライン上で

```
-fno-???
```

と指定すれば，環境変数上での設定を無効にすることができます。その他のオプションは，否定できません。オプションスイッチの詳細については，Vol. 2 を参照してください。

1)stdio.h などの標準ヘッダです。

2)笑って許してください。

3)Human68k ではコマンド文字数は，普通 255 文字に制限されています。

- "A"
  -fforce-addr の指定
- "E"
  このオプションスイッチを設定すると，**Human68k** 標準エディタ ed.x が扱える形式のエラーメッセージを生成する
- "F"
  -fomit-frame-pointer の指定
- "G"
  このオプションを指定した場合，コンパイラは作業用メモリをメインメモリ上で使いつくすとスーパーバイザモードに移行して，GRAM[4]を作業用メモリとして使用する。GRAM の内容は無条件に破壊するので注意が必要である
- "I"
  -finline-functions の指定
- "L"
  -fstrength-reduce の指定
- "M"
  -fforce-mem の指定
- "O"
  X68000 専用の最適化パスを許可する。最終的なコードのループ内部不変定数の移動を行う
- "P"
  この場合だけ他のオプションとは異なり，pea.l 0.w と clr.l -(sp) のコードの選択をする。前者が2バイト多くて，2クロック高速なコードである。デフォルトは pea.l 0.w となっている
- "S"
  -fstack-check の指定
- "T"
  コンパイラが GRAM を使いつくしたときに，マウスプレーンテキスト RAM を作業メモリ[5]として利用する
- "W"
  -Wall の指定

4)512K バイトあります。

5)256K バイトあります。

❖ DOSEQU

GCC が生成するアセンブラファイルは，デフォルトでは doscall.equ という **Human68k** のシステムコールを扱うシンボルファイルを使用します。デフォルト以外のファイルを使いたい場合に，ここにそのファイルネームを指定しておけば，コンパイラはそれをそのままアセンブラソースに出力します。このファイルは DOSCALL 関数拡張[6]のために使われます。

❖ SXEQU

環境変数 DOSEQU と同様に，SX–Window 開発モードで使われる SXCALL 関数[7]のためのシンボルファイルとして，デフォルトの sxcall.equ 以外のファ

6)第 2.4.7 節 (P.55) 参照。

7)第 2.3.4 節 (P.45) 参照。

イルを使う際に，そのファイルネームを指定します。

❖ **真里子，MARIKO**

X68000 版 **GCC** の拡張[8]を許可するための環境変数です。この設定がない場合の **GCC** は，ほぼ 100% 普通の 68000 CPU のための **GCC** としてふるまいます。詳しい拡張内容については，第 2.4 節 (P.48) の「一般的な拡張」を参照してください。

```
SET 真里子= ABCDEF
```

- "A"
  2 進ビット表現の拡張，日本語識別子拡張，割り込み関数の記述
- "B"
  asm("frame *reg*") の拡張，asm("extern *reg*") の拡張
- "C"
  ソースコードデバッガ対応
- "D"，"E"
  疑似統合環境の実現
- "F"
  コンパイル過程の表示

❖ **満里奈，MARINA**

疑似統合環境で使われるエディタを指定します。環境変数 PATH か path の指定するディレクトリにエディタの実行形式が存在すれば，フルパスで指定しなくてもファイルネームだけで起動することができます。

8)別の言葉では極悪改造ともいいます。

2.1.3 起動方法と書式

GCC は，コマンドラインから次のような書式で起動します。

```
gcc <ファイル名> [<オプションスイッチ>]
```

<ファイル名> には ".c"，".s"，".has"，".o" の拡張子をもつファイルが複数個指定できます。コンパイラドライバ gcc.x は各拡張子に応じて必要な処理を順次実行し，最終的に 1 つの実行ファイルを生成します。<オプションスイッチ> で特に指定がなければ，1 番最初に指定されたベースファイル名に ".x" を付加した名前の実行ファイルを生成します。

<ファイル名> も <オプションスイッチ> も指定されなかった場合には，書式を説明するヘルプメッセージが表示されます[9]。また <オプションスイッチ> は "-" で始まる文字列で，その順番は <ファイル名> も含めて任意に指定することができます。

9)ヘルプメッセージは，gcc.x が収められているディレクトリに gcc.hlp というファイルがなければ，画面には表示されません。

2.1.4 オプションスイッチ

10)詳しい機能については，Vol. 2を参照してください。

GCCコンパイラドライバには，次のようなオプションスイッチがあります[10)]。これらのオプションスイッチはコマンドラインから直接指定することも，先ほど説明した環境変数に設定しておくこともできます。

- "-a" ブロック単位でのプロファイラ
- "-ansi" ANSI違反の報告
- "-C" コメントを削除しない
- "-c" オブジェクトファイルの生成
- "-D" マクロの定義
- "-E" プリプロセッサ処理結果の出力
- "-f" 最適化の許可／禁止
- "-g" ソースコードデバッグ情報の生成
- "-I" インクルードパスの指定
- "-l" ライブラリの指定
- "-M" ファイル依存関係の出力
- "-MM" ファイル依存関係の出力
- "-mregparm" 引数をレジスタ渡しにする
- "-mshort" intを16ビットにする
- "-m68881" 68881用コードの生成
- "-O" 最適化の実行
- "-o" 出力ファイルネームの指定
- "-p" プロファイラコードの生成
- "-pedantic" ANSIに厳密に適合
- "-Q" バーボーズモード指定
- "-S" アセンブラソースの生成
- "-traditional" 伝統的なC言語仕様に準拠
- "-trigraph" trigraphシーケンスの認識
- "-U" マクロの削除
- "-v" コマンドラインの表示
- "-version" コンパイラのバージョン表示
- "-W" ワーニングの許可／指定されたワーニングの許可
- "-w" ワーニングの禁止
- "-as-symbols" アセンブラの最大シンボル数の指定
- "-cc1-stack" コンパイラスタック量の指定
- "-cpp-stack" プリプロセッサスタック量の指定
- "-fall-bsr" すべての関数をショートコールで指定[11)]
- "-fall-jsr" すべての関数をロングコールで指定[12)]
- "-fall-remote" 記憶クラスの固定化
- "-fall-text" テキストセクションですべてを出力する

11)X680x0 GCCでは，変更されています。

12)X680x0 GCCでは，廃止されています。

- "-fno-const-mult-expand"　定数乗法展開の禁止
- "-frtl-debug"　rtl の書き出し
- "-fscd"　ソースコードデバッグの生成
- "-fstrings-align"　文字列の偶数整合
- "-fstack-check"　スタックチェックコードの生成
- "-fstrings-nopcr"　文字列のプログラム相対禁止[13)]
- "-fstruct-strict-align"　構造体のパッキング
- "-ftext-report"　詳細なエラー報告
- "-fundump"　undump コンパイルの指定
- "-SX"　SX–Window プログラムモードの指定
- "-z-heap"　ヒープサイズの指定
- "-z-stack"　スタックサイズの指定

13)X680x0 GCC では，廃止されています。

2.2 ……………………………… GCC 拡張機能

GCC にはたくさんの拡張機能が用意されています。これらの拡張機能は，プログラムを書く際に，非常に都合がよいように合理的に拡張されています。ただし，

一般の C コンパイラでは GCC のほとんどの拡張機能は使えません

ので，移植性を考慮する場合は必ず #define されている __GNUC__ を利用するようにすべきでしょう。このセクションでは GNU オリジナルのドキュメントを参考にして，実際の X68000 でプログラミングする場合に，これらの拡張機能を有効に使用する方法を紹介していきます。しかし，GCC のすべての拡張機能にわたって解説するわけではありません[1]。

1)この他の拡張機能で，特に X68000 で有効なものはないと思います。

2.2.1 式の中の文と宣言

“{” と “}” で囲まれたブロックの先頭では，ブロック内でのみ使う変数を宣言することができます。これは変数のスコープを短くできるので，特に GCC では最適化の性質上，有効な変数の宣言方法ですが，さらに “{” と “}” で囲まれた，

ブロックを式の中で作る

ことができます。“{” と “}” で囲まれたブロックは文なので，それを式にするには “(” と “)” でその外側を囲みます。その式の値は “{” と “}” 内で最後に評価された式になります。ですから List 2-1 はエラーになります。なぜならばブロックの最終位置で，式が評価されていないからです。このとき，GCC は意味のよくわからないエラーメッセージを出力しますが[2]，要するに式でないものを評価させようとしたのがエラーの直接の原因です。

“{” と “}” で囲まれたブロックの最後で，式を評価させてください。C 言語では，List 2-2 で示す場合はすべて式の評価になります。

実用になる例として，char *strcpy (char *d, char *s) 関数を考えます。char *strcpy (char *d, char *s) 関数は，文字列をコピーするだけの非常に簡単な関数です。GCC でよく用いられるのは inline 関数[3]を用いた展開ですが，実際には List 2-3 のようなマクロのほうが有効です。

なぜならば X68000 GCC では，関数内で inline 宣言された関数が展開されると，スタックフレームを作成する LINK, UNLK 命令が必ず生成されてしまう

2)メッセージの和訳がまずいのですが。

3)第 2.2.10 節 (P.33) 参照。

からです[4)]。また inline 関数は使われた数だけ，その情報をその関数の呼び出しに備えてメモリ上に保管しなければならないため，コンパイラが使えるメモリ空間を減少させることにもなります。

4)GCC Ver. 2 ではこの問題は解消されています。

List 2-1 ● 誤った使用方法

```
1:  foo (int x)
2:  {
3:    int y = ({
4:        if (x > 0)
5:          5;
6:        else
7:          6;
8:            });
9:  }
```

List 2-2 ● 式の評価例

```
1:  "abcd";
2:  0;
3:  5;
4:  foo ();
5:  i,j,k;
```

List 2-3 ● マクロ化した strcpy ()

```
1:  #define strcpy (D,S)           \
2:  ({ char *_s = (S);             \
3:     char *_d = (D);             \
4:     while (*_d ++ = *_s ++)     \
5:       ;                         \
6:     (D);                        \
7:  })
```

また **GCC** 拡張機能を用いなくても，たいてい strcpy () 関数の戻り値は捨てられるので，普通の C コンパイラで用いる List 2-4 のようなマクロでも同じ働きのように見えます。ですが List 2-4 のようなマクロは，プリプロセッサによってマクロ展開されると式ではないため “,” 演算子で並べることはできませんし，“文” が置ける場所でないと strcpy () を記述することもできません。List 2-3 で示した例題の strcpy () を用いた場合，最適化を指定してコンパイルすると，**GCC** は List 2-5 に示すような，考えられる最短のコードをほぼ 100% 生成します。このようなマクロの欠点として，

- **デバッグ時にソースファイルとの対応がとりにくい (あるいはとれない)**
- **“{” と “}” で囲まれたブロック内部で宣言される変数と外部変数との衝突**

があげられます。たとえば，List 2-3 で strcpy (_s, _d); という呼び出しをするとエラーになるでしょう。**ANSI C** では “_” で始まる変数名はシステム予約変数ですから，たいていの場合はこういうエラー呼び出しは起きませんが，“_s” という変数がシステムで使われていないという保証はどこにもありません。

List 2-4● 通常のC言語でのstrcpy()マクロ

```
1:  #define strcpy(D,S)                \
2:  do { char *_s = (S);               \
3:       char *_d = (D);               \
4:       while (*_d ++ = *_s ++)       \
5:         ;                           \
6:  } while (0)
```

List 2-5● 最適化されたstrcpy()

```
1:          ...
2:          ...
3:          lea ?C0(pc),a1
4:          lea _buf,a0
5:  ?2:
6:          move.b (a1)+,(a0)+
7:          bne ?2
8:          ...
9:          ...
```

2.2.2 中間項を省略した条件式

? 演算子を用いた条件式では,

中間項を省略する

ことができます。たとえばList 2-6で示すようなプログラムの場合，この機能は便利です。List 2-6は**X68000 GCC**のソースコードに出てくる実際の例です。このソースファイルには2種類の環境変数があり，ここではそのどちらが定義されているのかあるいはどちらもされていないのかを判別しようとしています。そのため，一度 **getenv ("真里子")** を呼び出して評価しているにもかかわらず，同じ結果を返す **getenv ("真里子")** を再度呼び出しています。一般的にはこのような場合，一時変数を使わなければ呼び出しを一度だけにするのは難しいのですが，**GCC**の拡張機能を使えばList 2-7のように記述することができます。

List 2-6● 通常のC言語での記述

```
1:  ...
2:    {
3:      char *env
4:            = getenv ("真里子") ? getenv ("真里子"): getenv ("MARIKO");
5:      ....
6:    }
```

List 2-7● 中間項を省略した場合

```
1:  ...
2:    {
3:      char *env = getenv ("真里子") ? : getenv ("MARIKO");
```

```
4:        ....
5:    }
```

List 2-6の場合，getenv ("真里子") の呼び出しは副作用を伴わないので，const 属性[5)]を getenv () 関数につけておけば，最適化されて二度呼び出されることはなくなります。しかし，副作用がある関数を一度しか呼び出せない条件では，**GCC** 拡張機能を使わないと，一時変数の宣言なしには記述することはできないでしょう。たとえば，仮想的な2つの malloc () 関数のようなメモリ確保関数 malloc0 ()，malloc1 () を考えた場合，List 2-8のような記述を **GCC** 拡張機能を使わないで記述するのはめんどうです。

5)第 2.2.9.1 節 (P.32) 参照。

List 2-8 ● GCC 以外では記述が複雑になる例

```
1:      ...
2:    {
3:        char *mem = malloc0 (512) ? : malloc1 (512);
4:        ....
```

2.2.3 長さ0の配列

長さ 0 の配列，これは MS–DOS の著名コンパイラ MS–C でも **SHARP** 純正コンパイラ XC Ver. 2.1 でも実装されているサイズが 0 の配列です。XC ではマニュアルのどこを読んでもそのような記述はないのですが，なぜかヘッダにはそれらしき宣言が記述してあります。このヘッダ[6)]は，**GCC** に通すとエラーになるので書き換えが必要です。さて，この長さ 0 の配列は構造体を使って可変長のオブジェクトを管理する場合に便利です。List 2-9のようなデータ構造を用いる場合に長さ 0 の配列が使えないと，malloc () 関数でこのデータのメモリを確保するのがめんどうで美しくありません。

6)iocslib.h です。

List 2-9 ● 長さ0の配列の例

```
1:  typedef struct {
2:    int size;
3:    char buf[0];   /* XC, MS-C では buf[]; とサイズを書かない */
4:  } MS_BASIC_STR;
```

2.2.4 可変長配列

GCC では自動変数の配列確保で，

配列サイズが定数である必要がない

という画期的な拡張が施されています。この機能は麻薬的な便利さで，一度使うと，これができないコンパイラで文字列を操作する関数を記述するのはとても苦

痛になります。たとえば，filename から環境変数 temp に設定された一時ファイル用ディレクトリに filename のファイルを fopen () する場合，List 2-10 のように非常にエレガントに記述できます。配列 name_buf[] は，スタック上にそのエリアが確保されます。**X68000** の C で記述されたソースファイルからの実行形式は 64K バイトのスタックを用意していますから，普通に使うならばこの処理で十分です。この例では，malloc () を使うか MS-DOS のプログラムでよく見かける static な固定サイズ配列を使うのが一般的ですが，筆者としては，前者はともかく後者はいささか美しくないと思います[7]。

7)MS-DOSのCプログラムのスタックエリアの大きさは，想像を絶する少なさです。

List 2-10 ● 可変長配列の例

```
 1:  FILE *
 2:  testopen (char *filename)
 3:  {
 4:    char *env = getenv ("temp") ? : "./";
 5:    char name_buf[strlen (filename) + strlen (env) + 1];
 6:    strcpy (name_buf, env);
 7:    strcat (name_buf, filename);
 8:    return fopen (name_buf, "wb");
 9:  }
```

2.2.5　組み込み alloca ()

GNU プロダクトでは alloca () という名前の関数が頻繁に使われます。この関数は，その呼び出し関数内部にしかスコープがないという malloc () のような性格の関数です。**GCC** では，この関数は組み込み関数として実装されていますが，コンパイラ内部での処理は可変長配列宣言[8]とほとんど同じです。ただし，コンパイラは __builtin_alloca () に出会うつど，その引数を評価して必要なメモリ量を計算し，それをスタック上に確保します。そのため大規模な GNU プロダクト[9]では，普通に用意されている 64K バイトのスタックでは，

スタックがあふれて暴走する

ことがあります。この点に十分に注意して使用するようにしてください。List 2-11 は __builtin_alloca () 関数を用いてバッファを確保する場合の例です。

8)第 2.2.4 節 (P.27) 参照。

9)GCC も含まれます。

List 2-11 ● __builtin_alloca () を用いた例

```
 1:  #ifdef __GNUC__
 2:  #define alloca __builtin_alloca
 3:  #endif
 4:  void
 5:  foo (char *name)
 6:  {
 7:    char *buf = alloca (strlen (name) + 1);
 8:    ...
 9:    ...
10:  }       /* 確保した領域は明示開放しなくても捨てられる */
```

2.2.6 大域レジスタ変数

一般の C コンパイラでは，グローバル変数に対してレジスタを割り当てることはできません。ですが **GCC** では，

グローバル変数にレジスタを割り当てる

ことができます。ほとんどの関数で，共通にアクセスされる変数をレジスタに割り当てることによって，プログラムを高速化することが可能です。書式は List 2-12 で示すように，通常の変数宣言に組み合わせて，asm 文でどのレジスタに割り当てるかを宣言します。実際にレジスタ変数として使えるのはせいぜい 2 個程度です。

List 2-12● 大域レジスタ変数の宣言

```
1:  /* int はデータレジスタが適切 */
2:  register int bar asm ("d7");
3:
4:  /* pointer はアドレスレジスタが適切 */
5:  register int *foo asm ("a3");
```

大域レジスタとして割り当てられたレジスタは，関数で明示的に代入しないかぎりその値は不変で，コンパイラはそのレジスタを関数入り口で保存／復帰を行うようなコードは生成しません。このことは既存のライブラリがリンクできないということではありません。つまり，関数呼び出しで破壊されるレジスタはこの大域レジスタ変数にはできないため[10)]，ライブラリからこの大域レジスタ変数がアクセスできないというだけで，何ら問題はありません。

10)適切なエラーメッセージが出力されるでしょう。

この有効な利用方法の例として，TEX のプログラムがあります。TEX では，構成される関数のほとんどが共通の配列を参照しています。簡便のためにその配列を mem[] としておきます。List 2-13 のように，各関数で頻繁に mem[] をアクセスしてみましょう。

List 2-13● 大域配列変数のアクセス

```
1:  char mem[32768];
2:
3:  func (i)
4:  int i
5:  {
6:    func0 (mem[i],...);
7:      ....
8:  }
```

List 2-13 のような場合，大域レジスタ変数は有効です。この例では mem[] は配列ですから，そのままではレジスタに割り当てることができません。そこで List 2-14 のように，配列の先頭アドレスを常時レジスタに保持するようにします。これで mem[] へのアクセスはすべて a5 を通して行われ，大域レジスタ変数を使う前には，ほぼ関数ごとに生成されていた mem[] のアドレスをレジスタにロードする命令は，完全に消えてなくなります。TEX のソースファイルのように，

共通のデータにアクセスが集中的に起こるプログラム

では，この大域レジスタ変数は有効な手段です。

List 2-14● 大域レジスタ変数の活用

```
1: register char *mem asm ("a5");
2: char _mem[32768]; /* _ をつけておく */
3:
4: main()
5: {
6:   mem = _mem; /* 大域レジスタを初期化 */
7:   ...
8:   ...
9: }
```

2.2.7 レジスタ指定レジスタ変数

GCC では，通常の register 記憶クラス変数に割り当てるレジスタを，

特定のレジスタに指定

することができます。これを GCC の拡張 asm 文[11]と組み合わせて使った場合には，ほぼアセンブラなみの非常に低レベルのマシン操作を行うことができます。

書式は List 2-12 の大域レジスタ変数宣言と同じです[12]。宣言を行う場合には変数のスコープに十分な注意が必要です。なぜならば，

関数全体にスコープがあるような変数

にレジスタ指定を行うと，コンパイラの自動レジスタ割り当てに無理が生じるため，

効率の悪いコードが生成される

ことがあります。また，

必要以上の多数のレジスタ

をこの方法で割り当てると，コンパイラが自由に割り当てできるレジスタがなくなって，

コンパイラが異常終了する

ことがあります。これはコンパイラのバグではありませんので，そのような宣言を行わないようにしてください。

レジスタ指定変数の用途はほぼ asm 文に限られる[13]と思いますので，詳しい実例は第 2.2.11 節 (P.33) で紹介します。

11)第 2.2.11 節 (P.33) 参照。

12)第 2.2.6 節 (P.29) 参照。

13)何かほかに，これを使うもっともな理由があるでしょうか??

2.2.8 Constructor 表現

GCC は Constructor 表現[14]をサポートしています。Constructor 表現は“{”と“}”で囲まれた構造体の初期化要素構文に，キャストを施した形で実装されています。List 2-15 はその実例です。

14)オリジナルの英文では constructor expression と記載されています。

List 2-15 ● Constructor 表現の利用例

```
 1:  /* 構造体を作る */
 2:  typedef union {
 3:    struct {
 4:      short x;
 5:      short y;
 6:    } pos;
 7:    int x_y;
 8:  } point_t;
 9:
10:  void
11:  foo (void)
12:  {
13:    /* 拡張機能を使った呼び出し */
14:    call_p ((point_t) { 10, 20 });
15:  }
```

SX-Window のように，

一時的な構造体を関数に渡したいケース

が頻繁に発生するならば，この拡張は便利です。またこれを使えば，構造体配列の初期化も，各メンバをいちいち指定せずに一気に初期化できます。実際に使われている例としては，パソコン通信の SX-Window プログラム関係で有名な沖氏の SX-Clock[15]のソースファイルがあります。List 2-16 は筆者が作成したサンプル例です。

15)SX-Window 上の時計です。

List 2-16 ● SX-Window プログラムでの例

```
 1:  #include <sxdef.h>
 2:
 3:  LASCII buf[10];
 4:  void foo (LASCII *);
 5:
 6:  init (int index)
 7:  {
 8:    buf[index] = {
 9:            sizeof ("ぴんぽ〜ん") -1,
10:            "ぴんぽ〜ん"
11:    };
12:    foo (&(LASCII){
13:            sizeof ("LASCII タイプ文字"),
14:            "LASCII タイプ文字"
15:    });
16:  }
```

2.2.9 関数の属性宣言

GCC では関数に const と volatile の2種類の属性をつけることができます。この属性宣言は，コンパイラの最適化を助けるだけでなく，

無用の警告の出力を抑制する

場合にも役立ちます。

◆ const 関数

副作用のない関数および算術関数のほとんど[16]がこれに該当しますが，これらは List 2-17 のように宣言しておけば，共通部分式削除やループ内不変式移動の対象とされます。sin () の複数の呼び出しで引数が同じ値であれば，コンパイラは前の呼び出しの戻り値を利用するコードを生成しますし，引数がループ内で不変であれば，その呼び出しはループの外におかれて，その戻り値だけが利用されるでしょう。標準ヘッダ math.h に宣言してある関数を多用する科学計算プログラムを頻繁に使われるユーザは，これらの算術関数にすべて const 属性をつけておくべきです。

16)エラーが発生すると，errono が変更されますので，厳密には副作用がないとはいえません。

List 2-17 ● const 属性関数

```
1: /* 他のCとの互換性を考えて */
2: #ifdef __GNUC__
3: #define __const const
4: #else
5: #define __const
6: #endif
7:
8: __const double sin (double);
9: __const double cos (double);
```

1つ，注意点があります。

ポインタを引数としてその内容を調べる関数

には const 属性をつけてはいけません。文字列操作関数がその代表例です。また，

const でない関数を呼び出す関数

を const にしてはいけません[17]。

17)理由は考えてみましょう。

◆ volatile 関数

exit () 関数や abort () 関数あるいはユーザが作成した関数で，これらを呼び出すような，

呼び出し元へ復帰しない関数

があります。その際には，volatile と宣言します。この宣言は，関数の呼び出し後のジャンプ命令を生成しないといった類の最適化の意味もありますが，無用な未初期化変数の使用の警告を抑制するためにも有効です。

List 2-18● volatile属性関数

```
 1:  /* 他のCとの互換性を考えて */
 2:
 3:  #ifdef __GNUC__
 4:  #define __volatile volatile
 5:  #else
 6:  #define __volatile
 7:  #endif
 8:
 9:  __volatile void exit (int);
10:  __volatile void abort (void);
```

2.2.10 inline関数

関数を inline として定義することで，定義された関数を呼び出し部分に直接埋め込むことができます。第2.2.1節(P.24)で説明しましたが， inline 関数の展開を受けた関数は LINK，UNLK 命令が，必要なくても必ず生成されます。これらの命令は，68000ではクロック数の多い遅い命令ですので，マクロを使うか関数を使うかは，そのときの条件に応じて適切に切り替えるほうがよいでしょう。かなり大きめの関数でも inline 関数にできるので，strcpy () 関数のような簡単な関数[18]はマクロで，少し複雑でも呼び出しのオーバーヘッドを軽減したい場合は inline が望ましいと思います。またコンパイラは，inline 関数の呼び出しに備えて，そのすべてのコード生成情報をメモリ上に保持しておきます。その結果，コンパイラの作業メモリは減少します。このことをよく覚えておいてください。

inline 関数の前方参照はできません。inline 関数の実際の呼び出しの後にその実体を定義した場合には，その関数が static 関数であっても GCC はその実体を出力します[19]。なぜならば，その関数への参照は通常の関数と同様に， jsr 命令か bsr 命令で実際に呼び出しを行っているからです。

18)第2.2.1節(P.24)参照。

19)static な関数の呼び出しが全部展開できた場合には，GCC は実体を出力しません。

2.2.11 C言語オペランドをもつasm文

現在 **X68000** で最も多く利用されている **GCC** の拡張は，この asm 文でしょう。なぜならば，**GCC** の asm 文は XC での #asm のように，ソースファイルに含まれているアセンブラソースを，ただコンパイル結果に展開するだけの限定された機能ではないからです。asm 文の一般的な書式は List 2-19 のようになっていますが，これはあまりにも一般的すぎるので，少々わかりにくいかもしれません。また **X68000** で asm 文を活用するには，第2.2.7節(P.30)の「レジスタ指定レジスタ変数」の知識が必要になります。

List 2-19 ● asm 文書式

```
1:  asm ("インストラクション %op0 %op1..."
2:       : OUTPUT OPERAND_0,OUTPUT OPERAND_1,...
3:       : INPUT OPERAND_0,INPUT OPERAND_1,...
4:       :"破壊されるレジスタ名称 0","破壊されるレジスタ名称 1"...
5:      );
```

List 2-19 の OUTPUT OPERAND_0, INPUT OPERAND_0 などは,

" オペランド制約文字 " (C 言語での式)

で記述します。オペランド制約文字というのは, () で囲まれた式を評価代入する場合, その最終格納場所についての制限を記述する部分です。OUTPUT OPERAND_0 のオペランド制約文字の先頭は, 必ず "=" でなければなりません。またオペランド制約文字は CPU に依存しますので, ここでは話を 68000 CPU に限定[20]することにします。オペランド制約文字はたくさんありますが, 実際に asm 文で実用になるのは以下のものだけです。

20)他の CPU については GCC のソースファイルを読む必要があります。

- "d"　オペランドをデータレジスタに制約する
- "a"　オペランドをアドレスレジスタに制約する
- "r"　オペランドをレジスタに制約する
- "g"　上記オペランドを総括する[21]

21)実際には, 他の多数の制約文字も総括します。

オペランド制約文字の実際の働きについて見てみましょう。List 2-20 を見てください。int foo はメモリ上に存在する int の変数です。そして, d_test () 関数 の asm 文は次のようになっています (5 行目)。

```
asm ("テスト %0"::"d"(foo));
```

"%0" はコンパイラがオペランドを展開するフィールドで, asm 文に現れる INPUT OPERAND または OUTPUT OPERAND の順序で 0, 1, 2 と数値が増えていきます。このフィールドの数は INPUT OPERAND (または OUTPUT OPERAND) の数に一致する必要はありませんが, 展開を期待するフィールドの数値はオペランドの asm 文に現れる順序と一致していなければいけません。この例では, オペランドは INPUT OPERAND が 1 つ, foo だけです。

(foo) の前に現れている "d" がオペランド制約文字です。この例では "d" ですから, データレジスタに制約されます。コンパイラはこの制約に従って, 変数 foo を asm 文で評価する際にデータレジスタにロードする命令を発行します。List 2-21 のラベル _d_test: (4 行目) から 3 行が実際のコンパイル結果です。コンパイラは, 制約に従って変数 foo からデータレジスタ d0 に値をロードして, 次に asm 文の展開フィールド %0 に d0 を展開し, "テスト d0" という命令[22]を出力アセンブラコードに出します。

22)このような命令は, 68000 には存在しません。

List 2-20● オペランド制約文字の働き

```
 1:  int foo;
 2:
 3:  d_test ()
 4:  {
 5:    asm ("テスト %0"::"d"(foo));
 6:  }
 7:
 8:  a_test ()
 9:  {
10:    asm ("テスト%0"::"a"(foo));
11:  }
12:
13:  r_test ()
14:  {
15:    asm ("テスト%0"::"r"(foo));
16:  }
17:
18:  g_test ()
19:  {
20:    asm ("テスト%0"::"g"(foo));
21:  }
```

List 2-21● List 2-20のコンパイル結果

```
 1:          .text
 2:          .even
 3:          .globl _d_test
 4:  _d_test:
 5:          move.l _foo,d0
 6:  * APP ON (APP)
 7:          テスト d0
 8:  * APP OFF (NO_APP)
 9:          rts
10:          .even
11:          .globl _a_test
12:  _a_test:
13:          move.l _foo,a0
14:  * APP ON (APP)
15:          テスト a0
16:  * APP OFF (NO_APP)
17:          rts
18:          .even
19:          .globl _r_test
20:  _r_test:
21:          move.l _foo,d0
22:  * APP ON (APP)
23:          テスト d0
24:  * APP OFF (NO_APP)
25:          rts
26:          .even
27:          .globl _g_test
28:  _g_test:
29:  * APP ON (APP)
30:          テスト_foo
31:  * APP OFF (NO_APP)
32:          rts
```

同様に a_test () 関数では，制約に従ってアドレスレジスタ a0 に foo をロードしています。また，r_test () 関数のオペランド制約文字 "r" は，レジスタでありさえすればいい[23]ので，再度 d0 にロードされます。最後に現れる g_test () 関数の "g" というのは，“general operand” の “g” を表します。このオペランド制約文字は，アセンブラに現れるすべてのオペランド[24]が許され，アセンブラ命令のオペランドのアドレス形式が制限されているとき[25]に使います。

23)コプロセッサのレジスタは別の制約文字です。

24)浮動小数点タイプも許されます。

25)当然 GCC 自身が出力する命令の制限は GCC 自身が知っています。

アセンブラ命令によっては，INPUT OPERAND と OUTPUT OPERAND が共有されなければならないものがあります。たとえば lsl.l 命令がそうです。

List 2-22 では x を y の数だけ論理左シフトしようとしています[26]。この場合 INPUT OPERAND としては x と y の2つ，OUTPUT OPERAND は x です。普通に考えればこの asm 文は何ら問題ないように見えますが，複雑な関数の中でこのような asm 文が現れると，プログラマが予想もしなかった不都合が発生します。INPUT OPERAND に現れる変数 x と OUTPUT OPERAND に現れる変数 x が同じ場所にあるとはかぎらないからです。List 2-22 のコンパイルで，最悪の結果になったのが List 2-23 です。やっかいなことに List 2-22 のような場合，たいていはちゃんと動いてしまいます。しかしこれは本当にたまたま動いていただけで，最適化のオプションを変更したり，ソースファイルの改変などでレジスタの割り付けが変化したら，動かなくなることもあります。

26)これは asm 文を使う必要はないのですが。

List 2-22 ● 誤った asm 文の例

```
 1:  int bar;
 2:  int
 3:  foo (int x,int y)
 4:  {
 5:     ...
 6:    asm ("lsl.l %1,%0":"=d"(x):"d"(y),"d"(x));
 7:    bar = x;
 8:     ...
 9:     ...
10:  }
```

List 2-23 ● 誤った asm 文での最悪の結果例

```
1:           move.l 12(a6),d2  * y を取り出す
2:           move.l 8(a6),d0   * x を取り出す
3:           move.l d0,d1      * x を d1 にたまたまコピー
4:           ...               * 何か他の処理をする (d0,d1 は壊さなかった)
5:   * APP ON (APP)            * 入力としては d0 だけど
6:           lsl.l d2,d1       * 出力として d1 を使った
7:   * APP OFF (APP)
8:           move.l d0,_bar    * その後 d0 を bar へ
9:           ...
```

この他にも，asm 文が複数の命令で構成されている場合には注意が必要です。List 2-24[27]では，add.l 命令を 2 回行っています。コンパイル結果は List 2-25 ですが，これがプログラマの期待した結果ではないはずです。これは，foo が asm 文の中で OUTPUT OPERAND として 2 回変更されることを，GCC が知らないためにこのような結果になります。

27)これも asm 文でなくてもできます。

List 2-24● 複数命令の asm 文

```
1:  int foo;
2:  bar ()
3:  {
4:    asm (
5:    "add.l %1,%0\n\tadd.l %1,%0"
6:      : "=g"(foo)
7:      : "g"(foo));
8:  }
```

List 2-25● List 2-24 のコンパイル結果

```
1:          .even
2:          .globl _bar
3:  _bar:
4:  * APP ON (APP)
5:          add.l _foo, _foo
6:          add.l _foo, _foo
7:  * APP OFF (NO_APP)
8:          rts
```

こうした場合，オペランド制約文字に & を付加して記述します。この & が付加された場合，**GCC** は INPUT OPERAND をただ 1 回だけ評価して，**asm** 文に展開します。List 2-26 がその例で，List 2-27 がコンパイル結果です。たいていの場合，こちらがプログラマの意図した結果になります。

List 2-26● 複数命令の asm 文

```
1:  int foo;
2:  bar ()
3:  {
4:    asm (
5:    "add.l %1,%0\n\tadd.l %1,%0"
6:      : "=g"(foo)
7:      : "&g"(foo));
8:  }
```

List 2-27● List 2-26 のコンパイル結果

```
1:          .globl _bar
2:  _bar:
3:          move.l _foo,d0
4:  * APP ON (APP)
5:          add.l d0,_foo
6:          add.l d0,_foo
7:  * APP OFF (NO_APP)
8:          rts
```

オリジナルの **GCC** のマニュアルには「**asm** 文はコンパイラが知らない命令を使う場合に有効である」と記載されています。しかし **X68000** では，**GCC** が 68000 のインストラクションを下手なアセンブラプログラマより知っているくらいなので，知らない命令というのはかえってかぎられてきます。

そこで，有効な活用方法の手助けになるように，**asm** 文の実用例をいくつかあ

げておきます。**X68000**で最も有効なasm文の利用方法は，IOCSコールをプログラムから直接発行することでしょう。ここでは，IOCS $19 FNTGETをマクロで実現することを考えます。IOCS $19 FNTGETの仕様は次のとおりです。

Table 2-1● IOCS $19 FNTGETの仕様

IOCS $19	FNTGET	漢字パターンの取得
入力	d1.l	漢字コード
	a1.l	データバッファ
出力	d0.l	不定(破壊)
	d1.l	不定(破壊??)
	a1.l	不定(破壊??)

これをマクロでコーディングしたものがList 2-28です。このFNTGET ()は**GCC**拡張機能をふんだんに取り入れたコーディングとなっています。

- 2～4行
  IOCSが使うレジスタをレジスタ指定変数で確保する。外部変数との干渉を避けるために“_”を先頭に使う変数を用いる
- 5～7行
  レジスタ変数をパラメータで初期化する。“-O”オプションで最適化を指定してある場合には，余計な移動命令が生成されることはない
- 8～11行
  実際にIOCSを発行するアセンブラコードを，アセンブラソース出力ファイルに書き出させる
- 12行
  引数BUFを評価させて，FNTGET ()が関数のように記述されてもエラーにならないようにする

重要なのは，9行目の“:/* NO output */”です。asm文で書式List 2-19で示されているOUTPUT OPERANDの部分が省略された場合，そのasm文は最適化で移動されたり削除されたりはしません。このととは，ぜひ覚えておいてください。

List 2-28● asm文FNTGETマクロ

```
 1:  #define FNTGET (CODE,BUF)                              \
 2:  ({ register int _d0 asm ("d0");                        \
 3:     register int _d1 asm ("d1");                        \
 4:     register unsigned char *_a1 asm ("a1");             \
 5:    _d0 = 0x19;                                          \
 6:    _d1 = (CODE);                                        \
 7:    _a1 = (BUF);                                         \
 8:    asm ("trap #15"                                      \
 9:          :/* NO output */                               \
10:          : "d"(_d0),"d"(_d1),"a"(_a1)                   \
11:          : "d0","d1","a1");                             \
12:    (BUF);                                               \
13:  })
14:
15:  extern unsigned char buf[];
16:  void
```

```
17:  test (void)
18:  {
19:      /* L'あ' は XC ではコンパイルできません */
20:      FNTGET (L'あ', buf);
21:  }
```

次に IOCS $11，コントラストの設定を考えてみます。仕様は次のようになっています。

Table 2-2● IOCS $11 CONTRAST の仕様

IOCS $11	CONTRAST	コントラスト設定
入力	d1.b	設定値
出力	d0.l	前の設定値
	d1.l	不定 (破壊??)

この IOCS コールでは，前の設定値が d0 レジスタに戻ってきます。そのために asm 文が List 2-28 とは異なっています。

List 2-29● asm 文 CONTRAST マクロ

```
 1:  #define CONTRAST (VAL)                        \
 2:  ({ register int _d0 asm ("d0");              \
 3:     register int _d1 asm ("d1");              \
 4:     _d0 = 0x11;                               \
 5:     _d1 = (VAL);                              \
 6:     asm ("trap #15"                           \
 7:          :"=d"(_d0)                           \
 8:          :"0"(_d0),"d"(_d1)                   \
 9:          :"d0","d1");                         \
10:     _d0;                                      \
11:  })
12:
13:  int last_val;
14:
15:  call_func ()
16:  {
17:    last_val = CONTRAST (10);
18:  }
19:
20:  call_proc ()
21:  {
22:    CONTRAST (10);
23:  }
```

List 2-30● List 2-29 のコンパイル結果

```
 1:          .even
 2:          .globl _call_func
 3:  _call_func:
 4:          moveq.l #17,d0
 5:          moveq.l #10,d1
 6:  * APP ON (APP)
 7:          trap #15
 8:  * APP OFF (NO_APP)
 9:          move.l d0,_last_val
10:          rts
11:          .even
12:          .globl _call_proc
```

```
13:  _call_proc:
14:          rts
```

OUTPUT OPERAND が asm 文に存在した場合，その asm 文は最適化の対象になります。**GCC** は，OUTPUT OPERAND が変更を受ける以外は asm 文に副作用はないと仮定します。最適化によって asm 文が移動されたり，場合によっては消えてなくなったりします。asm 文の消失は，List 2-29 で IOCS が返してくる前のコントラスト値である d0 の値を利用していない場合[28]に起こります。

28)ローカルな変数に代入するだけではムダです。実際に値を使わないと削除されます。

この現象を防ぐには，asm 文を asm volatile () と記述します。やみくもに volatile すればいいというものではありません。たとえば，算術関数を asm 文で浮動小数点ドライバを直接コールするマクロを volatile にするのは，最適化を妨げるだけです。最も望ましい活用方法は，同じ働きをするマクロを２つ用意しておき，戻り値を利用して関数的に用いる asm 文と，そうでない手続き的に用いる asm volatile 文を明確に分けて使うようにすることです。

List 2-31 ● loop 外に asm 文が移動される例

```
 1:  int foo;
 2:
 3:  int *
 4:  bar (void)
 5:  {
 6:    int i;
 7:    int *ret;
 8:    for (i = 0; i < 10; i++)
 9:      asm ("move.l %1,%0":"=d"(ret):"g"(&foo));
10:    return ret;
11:  }
```

List 2-32 ● List 2-31 のコンパイル結果

```
 1:          .globl _bar
 2:  _bar:
 3:  * APP ON (APP)
 4:          move.l #_foo,d1     *loop 外に追い出しされている
 5:  * APP OFF (NO_APP)
 6:          moveq.l #9,d0
 7:  ?5:
 8:          dbra d0,?5
 9:          ext.l d0
10:          move.l d1,d0
11:          rts
```

2.2.12 asm 文出力フォーマット

asm 文に記述するオペランドには，いくつかのフォーマット指定ができます。たとえば，アセンブラでアドレスレジスタに絶対アドレスを得る場合には，表記に #ADDR を使った movea.l 命令と ADDR を使った lea とが存在しますが，**GCC**

で asm 文内部に記述する場合は双方とも "g"(&ADDR) になります。ここで，オペランドの出力形式を変化させなければアセンブラでエラーになります。以下に asm 文で使われるフォーマット文字をあげておきます。

- %@
  (sp) をオペランドとして出力する。ただし，スタックの内容が変化したことをコンパイラは知らない
- %+
  (sp)+ をオペランドとして出力する。ただし，スタックの内容とスタックポインタが変化したことをコンパイラは知らない
- %-
  -(sp) をオペランドとして出力する。ただし，スタックの内容とスタックポインタが変化したことをコンパイラは知らない
- %a
  アドレスを出力する[29]場合に，“#” を付加しない

29)普通は，変数に割り当てられたアセンブラのラベル。

List 2-33 が実際のコーディング例で (この例は実用例ではありません。注意してください)，List 2-34 はそのコンパイル結果です。スタックポインタを操作するオペランドフォーマット文字には十分な注意をはらってください。コンパイラは，

スタックが変化したことはまったく知らない

のですから，不用意な使い方は即暴走につながります。

List 2-33● asm 文のフォーマット文字

```
 1:  int val;
 2:  void
 3:  operand_test (int arg)
 4:  {
 5:    asm ("move.l %0,%@"::"d" (arg));
 6:    asm ("move.l %0,%-"::"d" (arg));
 7:    asm ("move.l %0,%+"::"d" (arg));
 8:    asm ("move.l %0,%1"::"g"(&val),"a" (arg));
 9:    asm ("lea %a0,%1"::"g"(&val),"a" (arg));
10:  }
```

List 2-34● List 2-33 のコンパイル結果

```
 1:          .globl _operand_test
 2:  _operand_test:
 3:          move.l 4(sp),d0
 4:  * APP ON (APP)
 5:          move.l d0,(sp)
 6:          move.l d0,-(sp)
 7:          move.l d0,(sp)+
 8:  * APP OFF (NO_APP)
 9:          move.l d0,a0
10:  * APP ON (APP)
11:          move.l #_val,a0
12:          lea _val,a0
13:  * APP OFF (NO_APP)
14:          rts
```

2.3 ……………………………………SX–Window

このセクションでは，**X68000 GCC** のオリジナル拡張機能の 1 つである SX–Window プログラム専用の拡張について説明します。

2.3.1 SX–Window アプリ開発について

X68000 の標準ウィンドウ環境である SX–Window 上で動くアプリケーションを開発する場合，純正コンパイラである XC を使うと不便な点がいくつかあります。

- **リエントラントなプログラムを作成するときにソースファイル上で工夫が必要**
- **システムコール発行にライブラリを使わなければならないので，システムコールごとにオーバーヘッドが発生する**

これらを改善するために，**X68000 GCC** では SX–Window プログラムモードが用意されています。このモードでは，プログラマは SX–Window が要求するリエントラント性を意識する必要はまったくありません。普通にプログラムすれば，出力プログラムは完全にリエントラントになりますので，SX–Window 上ではプログラム空間はタスクごとに完全に共有され，メモリ効率が飛躍的に向上します。この拡張機能は全面的に **HAS**，**HLK** の拡張に依存しています。ですから XC に添付されているアセンブラやリンカでは，この機能を利用することはできません。

2.3.2 SX–Window 対応手法

SX–Window 上でリエントラントに作成されたプログラム[1]は，(プログラム本体は共有していても) タスクに与えられたワークメモリはタスクごとに別個です。そのワークメモリをスタックとグローバル変数に分割して，プログラムは利用することになります。同一プログラムが同時に実行されてもデータは別個であり，それらは互いに干渉しません。これがリエントラントの意味でもあります。

1) 実行形式ファイルに属性を示すヘッダがついています。

X68000 GCC では SX–Window に対応するために，大域レジスタ[2]を使った手法でリエントラントを実現しています。つまり，タスクごとに与えられたメモリの位置を **GCC** の大域レジスタにセットして，すべての変数をそのレジスタからのオフセットとしてアクセスするのです。この方法は別に目新しい手法ではなく，OS9/68000 では，すべてのプログラムをこの方法で作成しています。

2)第 2.2.6 節 (P.29) 参照。

単に **GCC** の大域レジスタを使うだけでは，XC において全変数を 1 つの構造体上に定義しておいて，リエントラントを実現する方法と基本的な違いはありません。実は **X68000 GCC** の SX–Window プログラムモードではレジスタ a5 を大域レジスタとして確保しておき，すべての変数は a5 からのオフセットでアクセスするというように拡張されているだけで，そのオフセットの実際の値の解決は全部アセンブラとリンカに任せています。ですから **HAS/HLK** の拡張なしでは，この SX–Window プログラムモード自体が成立しません。アセンブラレベルでのより詳しい説明は第 3.3.4 節 (P.89) に記載されています。

かなり難解なことを書きましたが，実際にプログラミングとデバッグを行う場合には「すべての変数が a5 からのオフセットでアクセスされる」ことだけを頭にたたきこんでおけば十分だと思います。これらの拡張機能は，実際にはいくつかの予約語を拡張することで実現しています。また， SX–Window で用いられる Pascal ライクな文字列形式を，簡便に記述できる拡張もしてあります。

2.3.3　拡張された記憶クラス

通常の `auto`，`static`，`extern`，`register` に加えて，

- `remote`
- `common`
- `relocate`

の 3 つの記憶クラス指定予約語が増やされています。また，関数については，

- `SXCALL`

が予約語として拡張されています。**ANSI C** を意識する場合には `__remote`，`__common`，`__relocate`，または `__remote__`，`__common__`，`__relocate__` をそれぞれ代わりとして使うことができます。

◆ remote

通常の変数はすべて a5 からのオフセットでアクセスされますが，そのオフセットが 64K バイトを超える大きさになる場合に，オフセットを 32 ビットで扱えるようにするための予約語です。この予約語が指定された変数は，すべてオフセットを 32 ビットとして扱いますので，多少アクセスが遅くなります。List 2-35, List 2-36 はそのアクセス方法の実例です。`remote` 宣言された変数のアクセスが，2 段階になっているのがわかると思います。

List 2-35 ● 拡張記憶クラスアクセス方法

```
1:  remote int foo;
2:  int bar;
3:  void func (void)
4:  {
5:    foo = 10;
6:    bar = 20;
7:  }
```

List 2-36 ● List 2-35のコンパイル結果

```
1:          .globl _func
2:  _func:
3:          lea _foo,a0
4:          moveq.l #10,d0
5:          move.l d0,(a0,a5.l)
6:          moveq.l #20,d0
7:          move.l d0,_bar(a5)
8:          rts
```

◆ common

通常の変数はリエントラントを保つためにa5相対ですが，common を指定された変数は通常の“.data”あるいは“.bss”に割り当てられます。したがって，この変数は同じプログラムで別のタスクを生成する場合には共有されます。うまく使えばおもしろいことができますが，下手に扱うとバグの巣窟となる可能性もあります。

◆ relocate

非常に特殊な記憶クラスです。SX-Windowの再配置可能メモリブロックハンドル __sx_relocate[3]を経由して，アクセスします。__sx_relocate はポインタへのポインタですから，アクセスは相当遅くなりますが，メモリブロックは再配置可能なのでメモリ的には有利になります。なお __sx_relocate は remote でない通常の変数として予約されていますが，GCC は実体を生成しないので，ユーザが明示して初期化定義をしておいてください。

3)これはGCCでの予約変数とでもいうものです。

List 2-37 ● relocate変数のアクセス方法

```
1:  relocate int foo;
2:  void func (void)
3:  {
4:    foo = 10;
5:  }
```

List 2-38 ● List 2-37のコンパイル結果

```
1:          .text
2:          .even
3:          .globl _func
4:  _func:
5:          move.l ___sx_relocate(a5),a0
6:          move.l (a0),a0
```

```
 7:         moveq.l #10,d0
 8:         move.l d0,_foo(a0)
 9:         rts
10:         .offset    0       !!単なる.offset命令である点に注意!!
11:         .globl _foo
12:  _foo:
13:         .ds.b 4
14:         .end
```

2.3.4 SXCALL 宣言された関数

SXCALL 予約語は SXCALL をユーザプログラムから直接発行するための予約語です。SXCALL をライブラリから呼び出すのに比べると，オーバーヘッドを軽減できるので有利になります。SXCALL 宣言された関数は，

プロトタイプで宣言することが必須

になります。コンパイラは，プロトタイプを参照してスタックに long で引数を積むのか， short で積むのかを判断します。まちがったプロトタイプ宣言は即暴走という結果になります。また，ユーザが SXCALL 関数の本体を定義することはできません。SXCALL 関数は関数ではありますが，その扱いは，

アセンブラソースでの単なるマクロ

にすぎません。ですから GCC の最適化で，関数のアドレスをアドレスレジスタに入れて間接コールする最適化もしません (できません)。また，SXCALL 関数をポインタで扱うこともできません。なぜならば，この関数にはアドレスがないからです。

List 2-39● SXCALL 関数の特殊性

```
 1:  SXCALL void FUNCTION (void);
 2:  void func (void)
 3:  {
 4:    FUNCTION ();       /* OK!! */
 5:  }
 6:
 7:  error_call ()
 8:  {
 9:    void (*func)();
10:    func = FUNCTION;  /* コンパイルエラーになる */
11:    func ();
12:  }
```

2.3.5 SXCALL 関数の戻り値

SXシステムコールは最大2つの戻り値を d0 および a0 に返します。d0 は，通常の C の関数で値を返すときに使われるレジスタですから問題ありません。問題になるのは，a0 で返してくる値[4]についてです。このことを解決するために“_SXCALLPtr” が 内部的に void * として宣言されていて，この値を参照することによって SXCALL 関数の戻り値の a0 が得られます。つまり関数の適切な位置で，この変数を参照することで a0 にアクセス可能となります。

4)さらにやっかいなのは，こちらの値の方がたいてい重要です。

List 2-40 を参照してください。WMOpen () 関数はウィンドウをオープンする SXCALL を発行しますが，通常の関数の戻り値である d0 はリザルトステータスで window * は a0 にセットされています。この値は，通常の関数の戻り値としては参照できないので，_SXCALLPtr を参照して得ます。この変数はプログラム上で参照位置を起点にして，最も直前に実行された SXCALL の返した a0 を保持しています。

List 2-40 ● SXCALL の戻り値 a0 へのアクセス

```
 1:  /* SXCALL 関数を定義したヘッダ */
 2:  #include <sxlib.h>
 3:
 4:  window *win;
 5:
 6:  winopen ()
 7:  {
 8:     ....
 9:     /* SXCALL を発行 */
10:     WMOpen (.....);
11:     /* 戻り値 a0 を得る */
12:     win = _SXCALLPtr;
13:     ...
14:  }
```

SXCALL 宣言された関数の呼び出し以外の式やステートメントは，この参照の前にあってもかまいません。しかし，List 2-41 のような場合で SXCALL 宣言された関数の呼び出しが実行されたり，されなかったりするときの _SXCALLPtr の値については保証されていません。

また不必要に参照を遅らせた場合には，出力コードの最適化にも影響します。できるだけ SXCALL 宣言された関数の呼び出しの直後で，この _SXCALLPtr の値を参照するようにコーディングしておいてください。

List 2-41 ● 危険な戻り値 a0 へのアクセス

```
 1:  #include <sxlib.h>
 2:
 3:  window *win;
 4:
 5:  winopen ()
 6:  {
 7:     ....
 8:     if (条件)
 9:       {
```

```
10:         /* SXCALL を発行 */
11:         WMOpen (.....);
12:         /* 戻り値 a0 を得る */
13:       }
14:     /* 条件を満たしていない場合ゴミが */
15:     /* win に代入される              */
16:     win = _SXCALLPtr;
17:     ...
18: }
```

2.3.6 Pascal 形式の文字列

SX-Window で用いられる LASCII 型文字列を直接設定できるエスケープシーケンスが拡張されています。新しく拡張されたエスケープシーケンスは "\@" です。このエスケープシーケンスは "\@strings" のように，文字列リテラルの先頭に指定するだけで有効になります。List 2-42 は正しい設定例です。

ANSI C で規定されている隣り合った文字列の結合では，結合される文字列のうち最初の文字列の先頭に，このエスケープシーケンスがあれば正しい設定です。文字列リテラルの最初の文字列でない位置や，文字列の先頭文字以外でのエスケープシーケンスは違法になります。List 2-43 がその誤った例です。

正しい宣言の場合には，コンパイラは最後の \0 (ヌル文字) を除いた文字列の長さを計算して，先頭に文字数の設定を行います。そのためエディタで文字数を数える作業は不用です。また "\@文字列"[1] のアドレスからの文字列は，そのまま C 言語の文字列になります。つまり，最後の \0 (ヌル文字) も存在します。さらに，255 文字を超えた設定はエラーになります。

List 2-42 ● LASCII 文字列設定

```
1:  #include <sxdef.h>
2:  /* LASCII 文字列 */
3:
4:  LASCII *str0 = (LASCII *) "\@LASCII 文字列";
5:  LASCII *str1 = (LASCII *) "\@結合""文字列";
```

List 2-43 ● 誤った設定

```
1:  #include <sxdef.h>
2:  /* LASCII 文字列 */
3:
4:  LASCII *str0 = (LASCII *) "LASCII\@文字列";
5:  LASCII *str1 = (LASCII *) "結合""\@文字列";
```

2.4 ……………………………………… 一般的な拡張

X68000 GCC には，**X68000** のハードウェアを利用するために便利であると思われる拡張が施してあります。これらは，環境変数“真里子”によってその拡張を使うかどうか，選択することができます[1]。この環境変数が設定されていない場合には，本セクションの拡張機能は使えません。

1)環境変数の設定については第2.1.2節(P.19)参照。

他のコンパイラでは，

これらの機能はまったく使えない

ので，ソースファイルのポータビリティは制限を受けます。そのため mariko_cc, MARIKO_CC, __mariko_cc__, __MARIKO_CC__ がプリディファインされます。ポータビリティを考慮する場合は，これらの機能を使う際にこのマクロをテストするようソースファイルに記述してください。

2.4.1 2進数ビット表現の拡張

環境変数“真里子”が示す文字列に“A”が設定してあると，整数定数の設定にXCと同じ2進数が使えるようになります。

List 2-44のように，普通の整数定数が記述できる場所に接頭子“0b”をもった2進数表現を使うことが可能です。また，プログラムの可読性を高めるためにList 2-45のように“_”で任意のビット位置で分離して記述できます。

List 2-44 ● 2進数ビット表現拡張(その1)

```
1:  int pattern[] =  {
2:    0b00000000,
3:    0b00111100,
4:    0b00111100,
5:    0b00110000
6:    0b00100000,
7:    0b00100000
8:  };
```

List 2-45● 2進数ビット表現拡張（その2）

```
1: int bits0 = 0b0000_1001_1100_1111;
2: int bits1 = 0b0000_1101_1100_1111;
3: int bits2 = 0b0000_1011_1100_1111;
```

2.4.2　日本語識別子の使用

環境変数“真里子”が示す文字列に“A”が設定してあると，C言語の文法規約で識別子が置ける場所にならば，漢字をおくことができます。たとえばList 2-46は“int”である“整数”という変数の宣言になります。

変数の宣言と同様に関数や構造体タグ，typedefなどの識別子にも漢字をそのまま使うことができます。注意が必要なのは，全角スペース文字も漢字の1つであるという点です。これは通常のスペースとは認識されません。通常使われるエディタが，この全角スペースを表示する機能をもたない場合にはソースファイルのデバッグが難しくなるので，十分に注意して希望していない位置に全角スペースが混入しないようにしてください。

List 2-46● 日本語識別子

```
1: int 整数;
```

2.4.3　SUPER()／B_SUPER()の特別処理

GCCでは，XCのライブラリ関数SUPER()関数とB_SUPER()関数を使う場合にスタック一括補正を禁止しないと，実行プログラムが暴走することがあります。その原因は，GCCが連続して関数を呼び出す際に呼び出しごとにスタックを補正せず，いくつかの関数をまとめて補正しているからです。回避するためには，この2関数を特別扱いにして，これらを呼び出す前にスタックを完全に補正してから呼び出し，暴走しないようなコードを生成します。スーパーバイザとユーザを行き来するプログラムでも，この関数名称で呼び出し／実行するかぎりは，-fno-deffer-pop[2]オプションを指定する必要はありません[3]。

2)Vol. 2のコマンドオプションを参照。

3)LIBCの同機能の関数についても同様に，-fno-deffer-popを指定する必要はありません。

2.4.4　割り込み処理関数

環境変数“真里子”が示す文字列に“A”が設定してあると，通常では記述ができない関数単体で，割り込み処理を行う関数を記述できます。X68000 GCCで

は **X68000** で発生するすべてのハードウェア割り込み，ソフトウェア割り込みを扱うことができます。

この拡張は MS-DOS のコンパイラで利用されている予約語を拡張する形式ではなく，**__builtin_saveregs ()** 関数を呼び出すことで，その関数全体が割り込み処理関数として処理されます[4)]。この関数は **GCC** の内部処理に影響を与えますが，実体があるわけではありません。したがって，この処理を行うライブラリがあるわけではないので注意してください。

4)この関数は，別機種の GCC では別の役割があります。

__builtin_saveregs () 関数は，0個か1個の引数をもつ関数としてソースファイルには記述します。この引数の数がその関数のふるまいを決定します。ただし同一の関数で，引数の数が異なる使い方をした場合の動作は不定です。

1. **引数が 0 個の場合**
   純粋な割り込み処理関数としてふるまいます。その処理関数が別の関数を呼び出さない場合は，関数が破壊するレジスタのみをスタックにセーブし，処理が終わったら **rte** で復帰します。関数呼び出しがある場合は，通常 **GCC** がセーブしないレジスタも，すべてスタックにプッシュして処理を実行します。外部に大域レジスタが宣言[5)]してあった場合は，そのレジスタは退避されません。しかし，**GCC** がこのレジスタをワークレジスタとして使うことはないので，特に大きな問題は生じません。明示的に **asm** 文を用いてレジスタ変数を宣言した場合は，ワーニングを出力して注意を促します。

5)第 2.2.6 節 (P.29) 参照。

2. **引数が 1 個の場合**
   引数をリターンアドレスとみなして処理が終わったら，すべてのレジスタを復帰して **rts** ジャンプします。つまりスタックにそのアドレスを格納して **rts** することによって，そこにエントリします。この処理は関数が記述されるごとに設定されますから，複数のジャンプ先を記述することができます。これによって **DOSCALL** 処理もフックして横取りすることが可能になります。さらに，引数が 0 (**NULL**) だった場合，その関数はすべてのレジスタを保存する関数としてふるまいます。この機能は，通常の関数にすべてのレジスタを退避するコードを付加したい場合に有効です。
3. **引数に __builtin_saveregs () 自身へのポインタを渡した場合**
   この場合，**GCC** は自動的に **rte** だけのダミー関数へジャンプするコードを生成します。この処理はある特定の場合だけ自前の処理を行わせて，それ以外の場合にはシステムのデフォルト処理に，処理を任せてしまいたい場合に有用です。

また割り込み処理には，通常の関数とは異なった多くの注意点があります。次に，代表的な注意点をあげておきます。

❖ 割り込み処理はスーパーバイザモードで動作

通常，C の関数はユーザモードで動作していて，システムの動作に支障を及ぼすような不当な動作はできないようになっています。たとえばポインタを用いて直接グラフィックを変更しようとしても，それはバスエラーで停止するでしょう。なぜならば，割り込み処理はスーパーバイザモードです。この

場合，そのアクセスがシステムに重大な損害を及ぼすようなものであっても停止しません。

❖ **割り込み関数から DOSCALL や IOCSCALL は通常発行できない**

割り込みは，いつどこで発生するか特定できない場合がほとんどです。たとえば画面に文字を表示している途中に割り込みがかかり，その処理でさらに画面を書き換えしようとすると，どうなるでしょうか？　結果は，神のみぞ知るといった事態が発生します。

❖ **デバッガではデバッグが難しい**

割り込み処理は“リアルタイム”処理です。たとえばタイマで時間を計測する処理を記述した場合に，その処理の中にブレークポイントをおくことはできません。なぜならばブレークポイントで停止すると，実際には時間が経過しているので，停止した次の瞬間に再度割り込みがかかって停止します。無限にこの過程が繰り返され，やがてスタックがあふれて暴走するからです。

また，ライブラリのいくつかは静的な変数を変更します。そのため，これらの処理を割り込み関数で呼び出した場合は正しい動作を期待できません[6)]。さらに，浮動小数点ドライバを呼び出しできるのかどうかは調査していないので，関数が浮動小数点関係の **gnulib** を呼び出す場合は十分注意してください。つまり，C言語で記述することのメリットは保守性能に関してだけで，安全性に関してはアセンブラで記述するのとまったく同じ条件だということです。通常，C のプログラムはユーザモードであり，68000 の場合は 86 系列に比べると安全性が極めて高くなっています。ヌルポインタのアクセスは 100% バスエラーで停止しますし，それによって OS 自体が破壊されることはまずありません。ところが，割り込み処理関数はスーパーバイザモードであり，68000 のもつ保護機能はほとんど無力になります。スタックもシステムスタックであることを忘れないでください。巨大なサイズのローカル変数の確保は暴走の可能性があります。

6)理解できない人は理解できるまで，この機能は使用しないでください。

そして **__builtin_saveregs ()** 関数を使ってジャンプする場合は，実際にフックする前に，必ずジャンプ先を設定してから割り込み処理を行うようにしてください。たいていの場合，この設定変数はグローバル変数で，初期値は 0 になります。そのため，番地 0 にジャンプすると悲惨な結果が待っています。フックしてから設定するのでは，割り込みがその直後で発生した場合，遅すぎるのです。ですから割り込み関数を，スタックチェックオプションでコンパイルするようなことはしないでください。

割り込みをプログラムで使う場合には，変数の揮発性に注意してください。**GCC** のような高度な最適化を行うコンパイラでは，**volatile** 宣言は極めて重要です。割り込み処理関数で変更するフラグなどは，必ず **volatile** 宣言をしてください。**GCC** は，

変数には予期しない変化は起こらない

ものとして最適化を行います。そして割り込み処理は，この“予期しない変化”を変数に対して引き起こす処理です。適切な **volatile** 宣言を行わなかった場合，プログラムが無限ループになってしまうことがあります。

List 2-47 はその典型的な例です。プログラマはフラグが割り込み処理でセットされて while ループを抜けることを期待しますが，最適化によってこのフラグはループ進入の前にレジスタにロードされて，そのレジスタをテストする命令に置き換えられてしまいます。結果的に，このプログラムは無限ループになってしまいます。

List 2-47● 無限ループ化する例

```
 1:  /* 割り込み処理でセットされるフラグ */
 2:  int warikomi;
 3:
 4:  void warikomi_wait ()
 5:  {
 6:      /* この while が無限ループになる */
 7:      while (!warikomi)
 8:        ;
 9:      warikomi = 0;
10:  }
```

◆ 割り込み処理ヘッダ

ソースファイル上に直接，独自拡張機能を記述するのはあまり望ましくありません。なぜならば，割り込み処理の拡張があまり美しくない方法で実装されているからです。本来，この拡張は“予約語の拡張”で行うべきものだったのですが，実装した当時は“予約語を拡張”することに抵抗があった[7]ために，このような奇抜な実装になってしまいました。ただし，多少でも可読性を高めるために，ヘッダを作成しておきました。

7)SX-Window 関係の拡張で，すでに大義名分は消失していますが。

interupt.h に割り込み処理関係のマクロが記述してあります。マクロには関数の復帰方法を記述するマクロと，引数と戻り値を設定するためのマクロの 2 種類あります。

- **IRTE ()**
  通常に rte する位置に記述する。通常の return と同じ感覚で記述できる
- **IRTS ()**
  関数は全レジスタを保存し，普通に rts 命令で復帰する。当然，関数は値を返すことはできない。また IRTE () と同時に記述することも不可能である
- **IJUMP (*addr*)**
  関数復帰のときは *addr* への jmp と同じ処理を行う。スタックには割り込み復帰アドレスなどが保存されているため，処理分岐先の *addr* の処理が rte で終われば，割り込み前のアドレスに正しく復帰する。IRTE () と同時に記述することはできない。システムデフォルト処理を行わないで復帰したい場合は， IJUMP_RTE () で行う
- **IJUMP_RTE ()**
  IJUMP (*addr*) と組み合せて使うマクロで，こちらを記述した場合は rte 命令に jmp する
- **PRAMREG (*var, reg*)**
  パラメータを受け取るレジスタネーム *reg* に変数 *var* を割り当てる

- `RETREG` (*var, reg*)
  戻り値を返すレジスタネーム *reg* に変数 *var* を割り当てる
- `SET_FRAME` (*reg*)
  フレームポインタレジスタを *reg* に割り当てる

これらのマクロで可読性に関しては少し改善されますが，割り込み処理の複雑さが改善されるわけではありません。割り込み処理を行う場合，慣れるまでは一度アセンブラソースを生成させて，自分の望む結果にコンパイルされているかどうかをチェックするようにしてください。

2.4.5　register 指定変数の拡張

割り込み関数の記述性を向上させるために，レジスタを指定した変数を大幅に強化しました。環境変数 “真里子” が示す文字列に “B” が設定してあると，この機能は有効になります。

❖ **asm("frame *reg*") の拡張**
DOSコールのサブルーチンを記述する場合，**a6** がパラメータポインタとなるため，**a6** をフレームポインタにはできません。そこで，**asm** 文でスタックフレームを **a3** ～ **a6** の任意のレジスタに設定できるように拡張しました。これはファイルの先頭か関数の先頭で，適当な変数を **asm** 文でレジスタ宣言することで有効になります。

❖ **asm("extern *reg*") の拡張**
IOCSコールのように，レジスタでパラメータを受け取る場合に使用するための拡張です。割り込み処理関数では，この宣言をされた変数を格納するレジスタは退避復帰されません。このため，値を格納して呼び出し元に値を返すことも可能です。

2.4.6　ダンプコンパイル

X68000 GCC には，ヘッダをプリコンパイルし，バイナリイメージにして，同一ファイルを何度もコンパイルする際に，コンパイル時間を短縮する機能があります。ただし，プリコンパイルするヘッダには変数定義と **inline** でない関数定義を含んでいてはいけません。この機能は，

#pragma dump "*filename*"

で実現されています。*filename* には書き出すバイナリイメージのファイルネームを入力します。省略した場合には，**Cdump.bin** に書き出されます。このバイナリイメージはコンパイラのワークそのもので，場合によっては数 M バイトの大きさに達することもあります。

List 2-48 はサンプルとして記述したものです。ファイルネームは **win.c** とでもしておきます。まずバイナリイメージを作成するために，最適化を含めたすべてのオプションを指定しておき，アセンブラソースを作成します。

List 2-48 ● プリコンパイル例

```
 1:  #include <sxlib.h>
 2:  #pragma dump
 3:  /* あくまでサンプルです */
 4:  void
 5:  foo (window *w)
 6:  {
 7:    if ((WMActive (),_SXCALLPtr) == w)
 8:      {
 9:        bar0 ();
10:        bar1 ();
11:      }
12:  }
```

```
A>gcc -O -S win.c
Write total size = 825148
コンパイルを中断します exec addr=94AFE
A>
```

この後，**win.s** を type コマンドで内容を確認します。

```
A>type win.s
* NO_APP
.include sxcall.equ

A>
```

上記のように，ファイルが空でない場合にはダンプコンパイルはできませんので，そのようなときはあきらめます。この場合はカラッポですから，ダンプコンパイル可能です。一般的にヘッダの中に変数が宣言されていない場合には，この例のような空のファイルが出力されます。

その後，メモリ配置に変更がいっさいない場合は，次のように再度コンパイルすることができます。

```
A>gcc win.c -O -S -SX -fundump-Cdump.bin
A>
```

このように **sxlib.h** のコンパイル時間を節約して，**win.c** を何度でもコンパイルすることができます。ただし常駐プログラムが変化したりして，**Cdump.bin** に書き込まれているアドレス情報が次のコンパイル時に異なった値になっている場合には，コンパイルを実行しません。このダンプイメージをアドレスに依存しない

形式のファイルに変換するバッチファイルが，インストーラによってインストールされています。gccdump.bat がそのバッチファイルです。このバッチファイルを使用して，リロケート可能なイメージを作成するときには，書き出されるダンプイメージの約 3 倍のディスク容量が必要となります。作成時にディスク容量が不足すると，正しいリロケートができませんので，十分注意してください。

また，“-fundump-*filename*” がオプションスイッチとして指定された場合は，**#pragma dump** "*filename*" に出会うまで，マクロの定義以外はソースファイルを読み飛ばします。コンパイルは **#pragma dump** "*filename*" の次の行から開始されます。ダンプコンパイルを実行する際は，この読み飛ばしが行われるということを覚えておいてください。

筆者は，このダンプコンパイルにどれほどのメリットがあるのかよくわかっていません。コンパイル時間も重要ですが，高速な SCSI ディスクを装備していない **X68000** の場合には，リロケートデータのリード時間だけでもけっこう時間を消費しますので，それほど大きなメリットはなさそうに思われます。

2.4.7 DOSCALL 関数

SX–Window 開発モードでの SXCALL と同様の関数として，DOSCALL 関数が拡張されています。この拡張は SXCALL 関数と同様に，**Human68k** のシステムコールを直接発行する目的で行われています。この目的のために予約語として，

- DOSCALL
- _DOSCALL
- _DOSCALL__

が新しく追加されています。DOSCALL 関数の内部的な扱いは SXCALL 関数[8]と同一ですが，次の 2 点が異なっています。

8)第 2.3.4 節 (P.45) 参照。

- **アセンブラでの呼び出しマクロが異なる**
- **特殊変数 _SXCALLPtr[9]が予約されていない**

9)第 2.3.5 節 (P.46) 参照。

この拡張を使えば，**Human68k** のほとんどの機能をライブラリに頼らないで実行することができます。

2.4.8 疑似統合環境

村上氏のご協力により，コンパイラドライバに疑似的な統合環境をもたせることができました。環境変数 “真里子” と “満里奈” でこの機能を制御します。“満里奈” には，各自使用しているエディタを拡張子も含めて設定してください。デフォルトは EM.X です。エディタには EM.X が起動された場合を除いて，エラータグファイル mariko.err が渡されますので，それを使ってタグジャンプしてくだ

さい。環境変数“真里子”に“D”を設定します。“D”と同時に“E”を設定すると，ワーニングもエラー扱いになり，自動的にエディタが起動するようになります。

Human68kでの標準エディタ ed.x で，このタグジャンプコンパイルを具体的に説明します。List 2-49 はエラーが含まれているプログラムです。これを test.c とします。

List 2-49● タグジャンプサンプル

```
 1: #include <stdio.h>
 2: main (int argc, char **argv)
 3: {
 4:   if (argc == 3)
 5:     {
 6:       FILE *infile;
 7:       FILE *outfile;
 8:       int c;
 9:       infike = fopen (argv[1] ,"r");
10:       if (!infile)
11:         {
12:           fprinf (stderr,
13:                   "ファイルをオープンできません %s \n",
14:                   argv[1]);
15:         }
16:       outfile = fopen (argv[2] ,"wb");
17:       if (!outfile)
18:         {
19:           fprinf (stderr,
20:                   "ファイルをオープンできません %s \n",
21:                   argv[2]);
22:         }
23:       while ((c = fgetc (infile)) != EOF)
24:         fputc (c, outfp);
25:     }
26: }
```

10)環境変数の設定については第2.1.2節(P.19)を参照。

test.c プログラムをコンパイルする前に，**X68000 GCC** の環境変数[10]をセットします。**GCC** のエラーメッセージは，デフォルトでは ed.x が受けつける形式ではないので，まずそれを **GCC** に通知する必要があります。

```
A>SET 真里子= D
A>SET 満里奈= ed.x
A>SET GCC_OPTION= E
A>gcc test.c -O
```

このように設定してからコンパイルすると，

```
test.c: In function main:
test.c      9: Error   : 'infike' は未宣言です
test.c      9: Error   : (未宣言識別子は出現した関数ごとに一度しか
報告しません)
test.c     24: Error   : 'outfp' は未宣言です
```

このような表示がなされた後，ed.x がタグファイルを与えられて起動します。修正したいエラーを表示している行の先頭にカーソルを移動して，ESC キーをタイプした後，v キーをタイプします。ソースファイルが読み込まれてエラー位置にカーソルがジャンプします。エラーを修正したら ed.x を抜けてください。自動的にソースファイルが再度コンパイルされます。エラーが複数存在している場合は，修正によって行が移動することがあるので，後ろのエラーから修正すると効率的です。1 つのエラーを修正して，ESC キーの後で v キーをタイプするとタグファイルに戻るので，新しい修正部分をカーソルで指定します。同様の操作を繰り返せば，次々とエラーを修正できます。

2.4.9　プロファイラ

このプロファイラ機能は，オリジナル **GCC** のプロファイラ機能を改造して **X68000** 用にインプリメントされた機能です。C 言語のプログラムを作成してそのパフォーマンスを検証する場合に，どの関数で時間を消費しているか，また関数のどの部分で時間を消費しているかを，詳細に報告させることができます。プロファイラ機能を有効にするオプションは "-p" と "-a" オプションです。

これらのプロファイラ機能が指定された場合，コンパイラは関数や構文単位について時間を測定するための余分なコードを生成します。この余分なコードはプロファイラライブラリを呼び出すためのコードで，プログラムが終了したとき，呼び出し元に制御が移る前に，時間情報を標準出力に出力します。"-a" オプションで関数内部ブロックごとにプロファイラを指定した場合には，同時に "-S" オプションで，アセンブラコードのどの部分がソースファイルのどの部分に該当するのかを知っておく必要があります。さもないと，出力される情報を見ても何の意味ももちませんので注意してください。

2.4.10　#pragma の拡張

X68000 GCC では **#pragma** が大幅に拡張されています。オリジナルの **GCC** では **#pragma** はただ単に無視されるだけですが，**X68000 GCC** では，通常はコンパイラドライバのオプションとして指定する最適化オプションとコード生成の方法を指定するオプションをソースプログラムの任意の位置で有効にしたり無効にしたりすることが可能です。書式は

```
#pragma String [on | off]
```

です。*String* には，コンパイラのコード生成を指定するものと最適化の方法を指定するものがあります。

❖ コード生成について指定するもの

- `regparm`

  on が指定されると，一時的に関数呼び出しや引数の受け取り方法がレジスタを経由しての方法に変更されます。
- 68881

  on が指定されると，一時的に浮動小数点の計算にコプロセッサを用いた計算方法に変更します。
- `short`

  on が指定されると，デフォルトの `int` のサイズを一時的に 16 ビットに変更します。

これらの on，`off` 指定はプログラムの位置で可能ですが，関数の内部で指定した場合には，安全に動作する保証はありません。ですが，十分に C 言語に詳しく生成された機械語の動作を理解できる場合には，かなりの自由度で任意の位置にこのプラグマを置くことができます。

❖ 最適化の方法について指定するもの

- `strength-reduce`
- `combine-regs`
- `omit-frame-pointer`
- `keep-inline-functions`
- `inline-functions`
- `defer-pop`

これらは，コンパイラドライバに対して指定する最適化オプションの機能と同じ機能を一時的に on，`off` することができます。ただし，最適化は関数の始めから終わりを単位として実行しますので，関数の途中でこれらを何度 on，`off` しても，関数の定義が終わる “}” をコンパイラが処理した時点での設定が有効になります。

2.4.11 その他の拡張

特に大きな拡張と呼べるものではありませんが，次の 3 点のような記述がプログラムに含まれているとき，**X68000 GCC** ではワーニングが出力するようにしてあります。

❖ `if` 文での定数代入

List 2-50 のように，条件式で定数を変数に代入すると警告します。

List 2-50● 意味のない if 条件での代入式

```
1:  void foo (int x)
2:  {
3:    if (x = 5)
4:     {
5:        ....
6:        ....
7:     }
8:  }
```

❖ 32K バイトを超えるローカル変数

関数で，ローカル変数のトータルサイズが 32K バイトを超えた場合にワーニングが発生します。“小さな親切，余計なお世話” 的なワーニングですが，ないよりは変な暴走が回避できると思われるので増設してあります。

❖ 関数引数で変換が起こった場合

プロトタイプと異なったタイプの引数を関数に与えた場合に，何番目の引数でそれが起こったのかを明示するように拡張してあります。

2.5 ROM化について

この開発ツールで作られたプログラムはROM化して，68000をCPUとする組み込み制御装置などのプログラムとして使うことができます。**GCC**が出力するコードは，ROM化を行ううえで問題となる部分はありません。拡張子`X`をもつ**X68000**の実行形式は，実行番地に依存しないソフトウェアリロケータブルな形式です。付属ディスクには，この**X68000**の実行形式から純粋な絶対アドレスバイナリを作成するツール`cvm.x`とそのソースファイルが添付してあります。

ROM化する場合，通常ではC言語の外部変数の初期化ができません。OS9/68000のように，外部変数を初期化するOS上で稼働するROMならば問題はありません[1)]。通常のスタンドアロンなプログラムを作成する場合には，次のようなことに注意すれば，比較的容易にROM化することができます。

1)OS9/68000自体も，ROM化される必要があります。

- **スタックの設定などを行う初期化部分は自前で用意する**
- **文字列リテラルをポインタなどで変更しない**
- **C言語の外部変数初期化構文を使わない**
- **外部変数を参照するライブラリ[2)]を使わない**

2)著作権の問題もあります。

機械の構成によってRAMの番地が変化するようなマシンをターゲットにする場合には，**GCC**のSX-Windowプログラムモードを使ってプログラミングしましょう。a5レジスタにRAMの適切な位置を設定する部分を変更するだけで，RAMの位置をまったく別の位置に移動させることもできます。

2.6 ……………………………GCC の非互換性

GCC のオリジナルマニュアルでは UNIX の PCC と比較して記述してありますが，本書では X68000 の XC と比較するように書き改めておきます。非互換性はほとんど“伝統的な C コンパイラ”と ANSI C 規格との相違に起因します。ですから“-traditional”オプションで，GCC に他の C コンパイラと同様にふるまうように指示することで，それらのほとんどを回避することができるようになっています。

❖ **文字列定数**

GCC では通常 read–only です。いくつかの“同一文字列”はまとめられて 1 つコピーされます。この結果，mktmp () 関数は文字列を引数に使用することができなくなります。なぜなら mktmp () 関数は，その引数のポインタが指している文字列をいつも変更するからです。

解決策は，文字列の代わりに char-array 変数を文字列で初期化して使うことです。もしこの方法が不可能ならば，“-fwritable-strings”オプションを使ってください。

❖ **文字列リテラル中でのマクロ**

GCC は文字列リテラルの中に現れるマクロの引数を展開しません。たとえば #define foo(a) "a" はマクロの引数“a”が何であっても "a" に展開します。

“-traditional”オプションは，そのようなマクロを non-ANSI C な形態のコンパイラと同様に扱うように指示します。

❖ **ブロック内での extern 宣言**

ブロック内部で宣言された extern 変数，関数のスコープは，そのブロック内部に制限されます。伝統的な C では，ブロック内部の extern 宣言は宣言以降のコンパイル単位内で有効です。

“-traditional”オプションは，GCC に“伝統的な”コンパイラと同じように，ブロック内部の extern 宣言を扱うように指示します。

❖ **typedef での結合**

typedef で，long とその他を結合できません (List 2-51 参照)。ANSI C はこれを許していません。なぜなら long は暗黙に型が int であるとされているからです。これは，C ソースのレベルではなく bison[1] に与える文法規則の違いによるため，対処はできません。

1)GNU の yacc です。

List 2-51 ● typedefの非互換

```
1:   typedef int foo;
2:   typedef long foo bar;
```

❖ **typedef 名の引数使用**

PCC では typedef 名を関数の引数に使えますが，**GCC** では使えません。これも上記と同様対処できません。

❖ **+=**

“+=” は **ANSI C** および **GCC** では 1 つのトークンです。“+␣=” はエラーになりますが，XC ではエラーになりません。これも上記と同様対処できません。

❖ **#if での制約**

GCC では，List 2-52 のようにつねに “偽” である **#if** ディレクティブに文字列が含まれるとエラーになります。

コンパイルオプション “**-traditional**” はこのエラーを抑制します。

List 2-52 ● プリプロセッサの非互換

```
1:  #if 0
2:    I'm not strings
3:  #endif
```

❖ **float を返す関数**

float を返す関数は，PCC では double に変換されます。**GCC** では float をそのまま返します。

もし同じにしたいのであれば，意味どおりに double を返すように宣言しなければなりません。XC でも **GCC** と同じです[2)]。

2)XC Ver.2.1 で確認。

❖ **構造体を返す関数**

GCC では，構造体を返す関数は，多くの **UNIX** のコンパイラとはまったく異なった方法で値を返します。1，2，4，8バイトの構造体は，int と同じくレジスタで返されます。それ以外の構造体は，呼び出し側が戻り値の格納用として指定した構造体のアドレスをレジスタで渡し，呼び出された関数側でそこに戻り値を格納して返します。PCC や XC では静的な領域を確保して，そこに値を返します。

GCC ではこの方法は使いません。なぜならば，その方法は遅いしリエントラントでないからです。もし同じ方法を使用したいならば，オプション “**-fpcc-struct-return**” を指定してください。しかし **ANSI C** 規定ライブラリには，構造体を返す関数が存在します。このような関数を含む場合，オブジェクトの混在リンクは悲惨な結果を招くでしょう。

◆ リンケージ

本書の開発ツールを使った場合には，基本的に XC とは異なった仕様の部分があります。それが変数のリンケージです。またこの開発ツールで作成したソースファイルが，XC では多重シンボル定義エラーでコンパイルできないことがあります。これはこの開発システムが，ほぼ **UNIX** にコンパチブルなリンケージ規則をもっているからです。

ヘッダとして List 2-53 を共通にもつ List 2-54 と List 2-55 をコンパイルすると，XC では変数 `foo` が多重定義になってしまいリンクできません。しかし，この開発ツールで作成した場合には，**HLK** がワーニングを出力するだけで実行ファイルが得られます。

ANSI C ではこれら両方とも認められていますので，一概に XC の処理がまちがいというわけではありませんが，**UNIX** のフリーソフトウェアを **X68000** で作成する場合には問題になることがあります。List 2-53 のヘッダで宣言されている変数は，“仮の定義” と呼ばれています。

```
int foo;
int foo;
```

このように，複数回 `foo` を宣言してもエラーにはなりません。なぜならば，これは “宣言” であって “定義” ではないからです。“定義” は変数の場合，初期値を設定することで行われます。この “定義” が同時にコンパイルされる一連のファイルの中で最後まで行われなかった場合は，初期値 0 の “定義” とみなされます。“定義” は同時にリンクされるソースファイルの中で，ただ 1 回だけ行われることになっていますが，**UNIX** ではこの “仮の定義” から “定義” への格上げがリンクの段階で行われます。XC では，List 2-54 のコンパイルが終了した時点で変数 `foo` が “定義” されるので，List 2-55 で “`common.h`” が `#include` されたことによって “定義” された `foo` と衝突してリンクできないのです。

List 2-53 ● 共通ヘッダ (common.h)

```
1:  int foo;
2:  int bar;
```

List 2-54 ● 変数 foo を定義

```
1:  #include "common.h"
2:  int foo = 1;
```

List 2-55 ● メインプログラム

```
1:  #include "common.h"
2:  main ()
3:  {
4:  }
```

2.7 ………… X68000 GCCでのプログラム

X68000 GCCは優れた最適化能力をもつコンパイラですが，記述の方法によっては多少出力コードが変化します。このセクションでは，**X68000 GCC**でプログラミングする際に“頭の隅に置いておく”べき配慮について触れておきます。

2.7.1 X68000 GCCだけの最適化

X68000 GCCで環境変数GCC_OPTIONに“O”が設定してあった場合には，**X68000 GCC**だけの最適化が行われます。この最適化は，アセンブラソースを出力する直前に行われています。

◆ LOOP最適化

68000用のオリジナル**GCC**では，変数に定数を代入する場合に，その値が-128から127であれば，いったんデータレジスタにmoveq.l命令で値をロードしてから変数に代入します。この目的のためにデータレジスタを1つ割り当てるのですが，レジスタ割り当てに余裕がある場合には，このレジスタがループ内不変として残ることがあります。これをループの外に追い出すことで，“4クロック×ループ実行回数”だけ実行時間が節約されます。List 2-56がその典型的な例です。これは，malloc ()関数で確保したバッファを1で埋めて，そのバッファを返す関数です。

List 2-57がコンパイル結果です。プログラム内で，わりあい頻繁に出現するループ内での定数代入が最適化されています。この最適化は，List 2-56のようにレジスタ割り当てに余裕がある場合にしかできません。

List 2-56● LOOP最適化

```
 1:  int *
 2:  all_1_alloc (int size)
 3:  {
 4:    int *ret = malloc (size);
 5:    if (ret == 0)
 6:      return 0;
 7:    else
 8:      {
 9:        int i;
10:        for (i = 0; i < size; i++)
```

```
11:         ret[i] = 1;
12:     }
13:     return ret;
14: }
```

List 2-57 ● List 2-56 のコンパイル結果

```
 1:         .globl _all_1_alloc
 2: _all_1_alloc:
 3:         move.l d3,-(sp)
 4:         move.l 8(sp),d3
 5:         move.l d3,-(sp)
 6:         jsr _malloc
 7:         move.l d0,d1
 8:         addq.w #4,sp
 9:         bne ?2
10:         moveq.l #0,d0
11:         bra ?1
12: ?2:
13:         moveq.l #0,d0
14:         cmp.l d0,d3
15:         ble ?8
16:         move.l d1,a0
17:         moveq.l #1,d2     * LOOP 外に追い出された命令
18: ?7:
19:         move.l d2,(a0)+
20:         addq.l #1,d0
21:         cmp.l d0,d3
22:         bgt ?7
23: ?8:
24:         move.l d1,d0
25: ?1:
26:         move.l (sp)+,d3
27:         rts
```

◆ 定数代入の最適化

いくつかの変数を同一の値で初期化する場合，そのデータの **sizeof ()** を大きいものから順に初期化していくと，効率よく処理します。なぜならば，いったん値がレジスタにロードされた場合には，その値を積極的に再利用する最適化が行われるからです。これは，プログラムを初期化する際によく出現します。

List 2-58 の **init0 ()** 関数と **init1 ()** 関数は，結果的にはまったく同じことを行う関数です。しかしコンパイル結果は，List 2-59 のように異なったアセンブラソースに展開されます。

List 2-58 ● 定数初期化

```
 1: char  character;
 2: short word;
 3: int   dword;
 4:
 5: void
 6: init0 (void)
 7: {
 8:   dword = 2;
 9:   word  = 2;
10:   character = 2;
11: }
```

```
12:
13:  void
14:  init1 (void)
15:  {
16:    character = 2;
17:    word  = 2;
18:    dword = 2;
19:  }
```

List 2-59● List 2-58のコンパイル結果

```
 1:         .text
 2:         .even
 3:         .globl _init0
 4:  _init0:
 5:         moveq.l #2,d0
 6:         move.l d0,_dword
 7:         move.w d0,_word
 8:         move.b d0,_character
 9:         rts
10:         .even
11:         .globl _init1
12:  _init1:
13:         move.b #2,_character
14:         move.w #2,_word
15:         moveq.l #2,d0
16:         move.l d0,_dword
17:         rts
```

明らかに init0 () 関数のほうが高速で小さなコードです。この最適化は32ビットのメモリへの代入値が -128 から 127 の範囲なので，初期化の際に，一度レジスタに格納することを利用しています。このように最適化を生かすには，

同じ値を代入する文を順番に並べる

ように記述することです。またデータタイプが異なるものが含まれる場合は，

sizeof () が大きいもの

から代入するように配慮すれば，よりよいコードが期待できます。

2.7.2 ソースファイル記述方法

ソースファイルは不必要に関数を外部に見せないように記述します。同一ファイル内でしか使わない関数はそのファイルの先頭で static 宣言をしておけば，その関数呼び出しが bsr 命令で呼ばれるため，サイズ面で有利な記述になります。また，“-finline-functions” オプションの選択もより有効になります。逆に main () 関数が冒頭に記述されていて，適切な関数宣言がないとか関数の呼び出し順序がトップダウンに記述してあるようなソースファイルは一番不利なコードを生成します。

また変数宣言はできるだけ積極的にブロックを作り，その冒頭で宣言して使うほうが，より高速なコード生成を期待できます。特に変数のアドレスを使う自動

変数を，頻繁に参照するのは非常に不利になります。**scanf ()** 関数のような関数を呼び出して，その結果を頻繁に参照利用する場合は，値を受け取ってもその値を直接参照せず，別の変数に代入するほうがより有利なコードを生成します。

2.8 ……………………………………バグについて

GCCは非常に洗練された高度な最適化を行うコンパイラです。著名な某コンパイラのように“安全でない最適化”は行わないので，その出力するコードは安心して使えます。しかし，X68000 GCCでは大幅な拡張をコンパイラに施したために，UNIXで稼働しているGCCに比べて信頼性が低下しているのは事実です。

その反面，このコンパイラでたくさんのフリーウェアなどが作成され，配布されていることもまた事実です。ですから“おや，コンパイラのバグかな？”と思われた場合は，このセクションを参照してください。

2.8.1 バグではないバグ

X68000 GCCで報告されたバグで，コンパイラのバグではないものについていくつかの事例を紹介します。

◆ プロトタイプ

今までで最も多数報告されたバグではないバグ報告が，プロトタイプ宣言でコンパイルエラーになることです。これは本当に頻繁に報告されていますので，パソコン通信をやっているユーザは何度か目にしたことがあるでしょう。

List 2-60は，プロトタイプ宣言されている関数が伝統的な関数定義でエラーになる例です。プロトタイプでは，`foo ()` 関数は引数を `char` と宣言しています。伝統的な関数定義で仮引数を `char` と宣言した場合，対応するプロトタイプでの仮引数のタイプは `int` です。しかしこのエラーは厳密すぎるようで，GCC Ver 2.2.2では，“`-pedantic`”オプションでワーニングになるレベルにまで緩和されています。

List 2-60● プロトタイプミスマッチ

```
1:  int foo (char);
2:  int
3:  foo (c)
4:    char c;
5:  {
6:    return c;
7:  }
```

◆ アセンブルエラー

巨大なソースファイルをコンパイルすると，アセンブラを通した段階でエラーになることがあります。特にUNIXから移植するプログラムの場合に，このエラーがよく発生します。これはX68000 GCC独自の最適化が原因となっています。

- **関数の引数に文字列を渡す場合には，プログラムカウンタ相対アドレッシングモードを使う**
- **static 宣言された関数は bsr で呼ばれる**
- **関数定義の後方参照は bsr で呼ばれる**

この2つの処理が原因となって，プログラムが大きすぎて相対番地に届かなくなるため，アセンブラがエラー終了するのです。これを回避するには，

```
-fstrings-nopcr
-fall-jsr
```

のどちらか，あるいは両方を使ってください。このほかには，シンボル数がアセンブラの許容数を超えてしまう場合があります。その場合には，

```
-as-symboles = <シンボル数>
```

で <シンボル数> を増やしてください。コンパイラドライバにコンパイルするファイルを1つだけ指定した場合には，作成されたアセンブラソースとオブジェクトファイルは削除されません。これらを修正してコンパイルを継続することも可能です[1]。

1)X680x0 GCCでは，この種類のアセンブルエラーは発生しません。

◆ コンパイラ異常終了

メインメモリが少ない環境で，グラフィックRAMやテキストRAMをワークに使った場合，コンパイラがバスエラーなどで異常終了することがあります。これはかなり危険なことです。異常終了する原因は，コンパイラワークを割り当てていることですが，これに対しては完全に対処しきれていません。“-cc1-stack”オプションでスタックの量を少し変化させると，コンパイルできることがあります。しかし，望ましいのはメインメモリを増やすことです。特にSX-Window関係のプログラミングでは，メモリを4Mバイト以上に増設しておくことをお勧めします。

また本体のハード改造でクロックアップを行っている場合には，メモリ化けに十分注意してください。GCCは，通常使わない最上位番地にスタックを設定して動きます。通常は正しく動いているように見えても，GCCのように普通は頻繁にアクセスしない領域を使うプログラムでは，欠陥を露呈することがあります。

◆ XCだと動く

よくあるバグ報告のなかに，“XCだとコンパイル実行できるのですが”といった類の報告があります。XCはANSI Cモドキのコンパイラで，完全にはANSI Cには準拠していません。細かな仕様の違いが，予期しない動作不良の原因になることがあります。また，GCCはXCに比べて文法が厳密です。昔，私が“痛い思い”をしたソースファイルをList 2-61にあげておきます。

List 2-61 ● XCでしかコンパイルできない

```
1:  void
2:  foo ()
3:  {
4:    /* , が余分にある */
5:    int i,j,k, ;
6:  }
```

このほかには，自動変数の初期化忘れがあります。**GCC**の場合は最適化を指定してコンパイルすると，自動変数はレジスタに自動的に割り当てられます。しかし，XCでは`register`宣言がないと，自動変数はスタック上に確保されます。変数の初期化を忘れた場合，XCではたまたまどのような条件でも確保したスタックが0になるといった“不思議な偶然”で動いていたプログラムが，**GCC**ではレジスタに残っていた単なるゴミを使ったために暴走するといったことがあります。

2.8.2　バグレポートについて

この**X68000 GCC**はNIFTY-ServeのFSHARP3でサポートしています。ただし個人範囲でのサポートですので，いわゆるコンパイラベンダーのようなサポートはできませんので，ご承知ください。このコンパイラを使っていて，

これはバグだ

と思われる現象に出会った場合は，まず本セクションの前半を読んで，該当現象かどうかを検討してください。検討した結果，

これは絶対バグだ

という結論に達し，幸運にもNIFTY-ServeのFSHARP3にアクセス可能なユーザは，次のような手順でご連絡ください。

1. NIFTY-ServeのFSHARP3で“バグ発見”と報告してください。
2. バグの発生するソースファイルを確定します。
3. そのソースファイルを「`gcc -E [ファイル名] > out.cpp`」としてください。
4. 私から連絡があったら`out.cpp`を電子メールしてください。

非常に困るのは，ただ“動作が変です”とか“XCだとちゃんと動きます”といったまったく意味のない報告です。それから，ソースファイルを送ってくださるユーザがいらっしゃいますが，たいていの方はインクルードしているヘッダをお忘れです。このようなソースファイルをいただいても，“黙って無視”するしか方法がありません。極端な表現をすれば，“添付したソースファイルでおかしくなります”が文面で，後はコンパイルできるソースファイルがついているほうが，いろいろ分析された結果を報告されるより，はるかに有効な情報なのです。

2.9 ……………………………………GCC の制限

このセクションでは，**X68000 GCC** のコンパイラとしての制限を文書化してあります。他のマシンで動いている **GCC** とは異なっている部分もあります。

❖ **識別子の長さ識別**

GCC 単体では外部リンケージ／内部リンケージともにメモリの許すかぎり，無制限です。**X68000** では **HAS** と **HLK** に依存します[1)]。大文字，小文字は区別されます。

1)第 3 章「HAS」と第 4 章「HLK」を参照してください。

❖ **`#include` とマクロ**

`#include` とマクロのネスティングレベルは 200 までです。**X68000** では事実上，メモリ残量と `config.sys` での `FILES` で制限されます。プリプロセッサはすべてのソースファイルをメモリ上で処理します。マクロの長さは残量メモリだけに制限されます。

❖ **文字列リテラルの長さ**

メモリの許すかぎり無制限です。ただし，`LASCII` 文字列は 255 文字までです。

❖ **関数の大きさ**

メモリの許すかぎり無制限です。実際には，大きな関数はヒープだけでなくスタックも大量に消費します。

❖ **初期化式の計算**

コンパイル時の定数計算でのオーバーフローは，“初期化式が複雑すぎる” というエラーになります。

❖ `long double`

`long double` は実装されていません。

X68000 では，**GCC** 自体の制限よりも，メモリ上の制限が先になりますので，実質上 “フリーメモリサイズによる制限” だけが制限といえるでしょう。

Chapter 3

X68000 HAS

High-speed Assembler (以下 HAS と略記) は，SHARP の X68000 用純正アセンブラ AS.X と上位コンパチブルである，68000 CPU 用のマクロアセンブラです。HAS では，純正アセンブラに本来備えられている標準的な機能に加えていくつかの拡張機能が追加されています。

3.1 ……………………………………… HASの概要

アセンブラは，アセンブリ言語で書かれたソースファイルをアセンブルしてオブジェクトファイルを生成します。このオブジェクトファイルをリンカによってリンクすることで，実行可能なファイルを生成することができます。

HASはこのアセンブラの1つで**SHARP**純正コンパイラXCに付属する純正アセンブラAS.Xと上位コンパチブル[1)]の機能をもっています。純正AS.Xに比べてアセンブル作業をより高速に行うことができ，さらにいくつかの拡張機能が追加されています。

1)基本的に，AS.Xでアセンブル可能なソースファイルはすべて，HASでもアセンブル可能です。

> 「*X680x0 Develop. & libc II*」に収録されているバージョンの**HAS**は，「*X68000 Develop.*」に収録のものに比べて，大幅な機能拡張が行われています。そのため本書では，機能拡張によって通用しなくなった内容に対する加筆修正を行っています。
>
> 文章中，「**X680x0 HAS**では」とは「*X680x0 Develop. & libc II*」に収録されているバージョンのHASを指し，「従来では」とは「*X68000 Develop.*」に収録されているバージョンを指します。

3.1.1 HASが使うファイル

HASは，アセンブルにあたって以下のファイルを使用します。この中には，アセンブラが参照するファイルやアセンブラ自身が作成するファイルも含まれます。

◆ 入力ファイル

次の2つのファイルはユーザが作成し，アセンブラに対する入力となるファイルです。

❖ ソースファイル

アセンブリ言語で書かれたファイルです。拡張子は通常“.s”を使用しますが，**HAS**で拡張されたオリジナルの機能を使用したソースファイルなどは，純正アセンブラAS.Xでアセンブルできないことを明示する意味で，拡張子に“.has”を使用することができます。拡張子が“.s”であるソースファイ

ルで，このオリジナルの機能を使用した場合には，純正アセンブラでアセンブルできないことを警告するために，ワーニングが発生します[2]。

❖ **インクルードファイル**[3]

これもソースファイルの一種ですが，その中で他のソースファイル中から .include 疑似命令[4]によって，参照されるファイルのことをいいます。インクルードファイルはソースファイルと同じディレクトリにおくこともできますが，他のディレクトリにまとめておいておき，環境変数 include でそのディレクトリを指定するか，またはアセンブルの際のコマンドライン上の "-i" スイッチによってディレクトリを指定することもできます。

2)X680x0 HAS では，拡張子に関わらずこのワーニングは発生しません。

3)インクルードファイルの拡張子は特に定められていませんが，'.h' は C 言語プログラムのヘッダファイルと混同する恐れがあるため，一般的には '.mac'，'.equ'，'.inc' などが使われているようです。

4)「ソースコードの挿入」を参照 (第 3.5.1 節 P.98)。

◆ 出力ファイル

次の 4 つのファイルは，入力されたファイルをもとにアセンブラ自身が作成するものです。

❖ **オブジェクトファイル**

ソースファイルをアセンブルすることによって得られるファイルです。ファイル名は，特に "-o" スイッチで指定しないかぎり，ソースファイル名の拡張子を ".o" に変更したファイル名となります。

❖ **リストファイル**

アセンブルリストを出力するファイルで，"-p" スイッチを指定することで作成されます。"-p" スイッチでファイル名を指定しないかぎり，リストファイルはソースファイル名の拡張子を ".prn" に変更したファイル名で作成されます。

❖ **シンボルファイル**

ソースファイルの中で使用されたシンボル名に関する情報を並べたファイルで，"-x" スイッチを指定することで作成されます。"-x" スイッチでファイル名を指定しないと，シンボルファイルの内容は画面上に出力されます。

❖ **テンポラリファイル**

アセンブラ自身が使用する作業用ファイルです。通常，作業はすべてメモリ上で行われますが，メモリ容量が不足した場合に，一時的にこのファイルを作成することがあります。アセンブルが終了すると，テンポラリファイルは自動的に削除されます。

通常，テンポラリファイルはカレントディレクトリ上に作成されますが，環境変数 temp が設定されているときはそのパス上に，またアセンブラのコマンドラインで "-t" スイッチが指定されると，このスイッチによって指定されたパス上に作成されます。

3.1.2 HAS が使う環境変数

HAS が参照する環境変数には，以下のものがあります。

❖ include

標準的なインクルードファイル[5]がおかれているディレクトリをフルパスで指定します。

❖ temp

アセンブラがテンポラリファイルを作成するディレクトリをフルパスで指定します。

❖ HAS[6]

HAS がつねに使用する オプションスイッチを設定します。設定されたオプションスイッチは，**HAS** を実行するときにコマンドラインの最後に追加されます。

5)具体的には，IOCSコールやDOSコールの定義ファイルである，IOCSCALL.MAC，DOSCALL.MACなどです。

6)'HAS'は大文字のみで，小文字は受けつけません。

3.1.3 起動方法と書式

HAS は，コマンドラインから次のような書式で起動します。

```
HAS [<オプションスイッチ>] <ファイル名>
```

<オプションスイッチ> には，アセンブラの動作を制御したり，必要な情報を付加するオプションスイッチを指定します。オプションスイッチの指定は "-" から始めますが，"/"(スラッシュ) から始めることもできます。

<ファイル名> には，アセンブルを行うソースファイル名を指定します。ソースファイル名の拡張子が ".s" または ".has" である場合は，拡張子を省略することができます。その際，拡張子 ".s" と ".has" の両方のファイルが存在するときには，拡張子が ".has" であるファイルが優先されます。

<オプションスイッチ> の指定がまちがっていたり，<ファイル名> が指定されなかった場合には，書式を説明するヘルプメッセージが表示されます。

また，つねに使用するオプションスイッチをあらかじめ環境変数 HAS に設定しておくことで，アセンブルの際には必ずそのスイッチが指定されているものとすることができます。このとき，環境変数 HAS の 1 文字目に "*" を設定すると，オプションスイッチ指定キャラクタとしての "/" の使用を禁止することができます。たとえば，コマンドライン上で次のように実行したとします。

```
A>set HAS=-n -m20000
```

上記のように，環境変数 HAS に "-n -m20000" を設定しておくと，最適化禁止，最大シンボル数 20000 の状態でアセンブルします。

なお **HAS** のもつオプションスイッチの機能は，すべて純正アセンブラ AS.X の上位コンパチブルなので，**HAS** のファイル名 HAS.X を AS.X に変更して使用することで，純正アセンブラとまったく同じように使用できるようになります。

3.1.4 オプションスイッチ

HASには，以下のようなオプションスイッチがあります[7]。これらのオプションスイッチは，コマンドライン上から直接指定することも，環境変数HASに設定しておくこともできます。また**X680x0 HAS**では，機能拡張にともなっていくつかの新たなオプションスイッチが追加されています[8]。

- “-8”　シンボルの識別長の指定
- “-a”　絶対ショートアドレス形式対応モードの指定[9]
- “-d”　全シンボルの外部定義指定
- “-f”　リストファイルのフォーマット指定
- “-i”　インクルードファイルのパス指定
- “-l”　タイトル表示の指定
- “-m”　最大シンボル数の指定[10]
- “-n”　最適化の禁止
- “-o”　オブジェクトファイル名の指定
- “-p”　リストファイルの作成
- “-q”　クイックイミディエイト形式への変換禁止[11]
- “-r”　相対セクション命令の使用許可[12]
- “-s”　シンボルの定義
- “-t”　テンポラリファイルのパス指定
- “-u”　未定義シンボルの外部参照指定
- “-w”　ワーニングレベルの指定
- “-x”　シンボル情報の出力指定
- “-z”　**HAS**オリジナル機能へのワーニング禁止[13]

7)オプションスイッチの詳細についてはVol. 2を参照してください。

8)追加されたオプションスイッチについては，「X680x0 Develop. & libc II」を参照。

9)X680x0 HASでは，廃止されています。

10)X680x0 HASでは，変更されています。

11)X680x0 HASでは，廃止されています。

12)X680x0 HASでは，廃止されています。

13)X680x0 HASでは，廃止されています。

3.2 ……………………………アセンブリ言語の文法

このセクションではアセンブリ言語プログラムを構成する要素について，次の項目ごとに説明します。

- **ステートメントの書式**
- **識別名**
- **定数**
- **演算子**

解説中，**HAS** によって拡張された機能には HAS拡張 というマークがついています。この機能を使用したソースプログラムは，純正アセンブラではアセンブルすることができませんので注意してください。

3.2.1　ステートメントの書式

アセンブリ言語のプログラムは，ステートメントで構成します。ソースファイルは行を基本単位として，1 行には必ず 1 つのステートメントが記述されます。ステートメントは，以下の 4 つのフィールドから構成されています。

- **ラベルフィールド**
- **オペレーションフィールド**
- **オペランドフィールド**
- **コメントフィールド**

フィールドは行の先頭からこの順に並び，それぞれのフィールド間は，1 文字以上のスペース (またはタブコード) で区切ります (Fig. 3-1)。また，各フィールドはいずれも省略することができます[1]。

1)ただし，オペレーションフィールドの命令がラベルフィールドやオペランドフィールドを必要とするものであれば，それらを省略することはできません。

◆ ラベルフィールド

ラベルフィールドには，任意のカラムから始めて “:” で終わるシンボル，または，第 1 カラムから始めてスペースまたはタブコードで終わるシンボルを書きます。このシンボルには，ラベルとしてアドレスが定義されたり，アセンブラ疑似命令によって定数やマクロなどが定義されたりします。

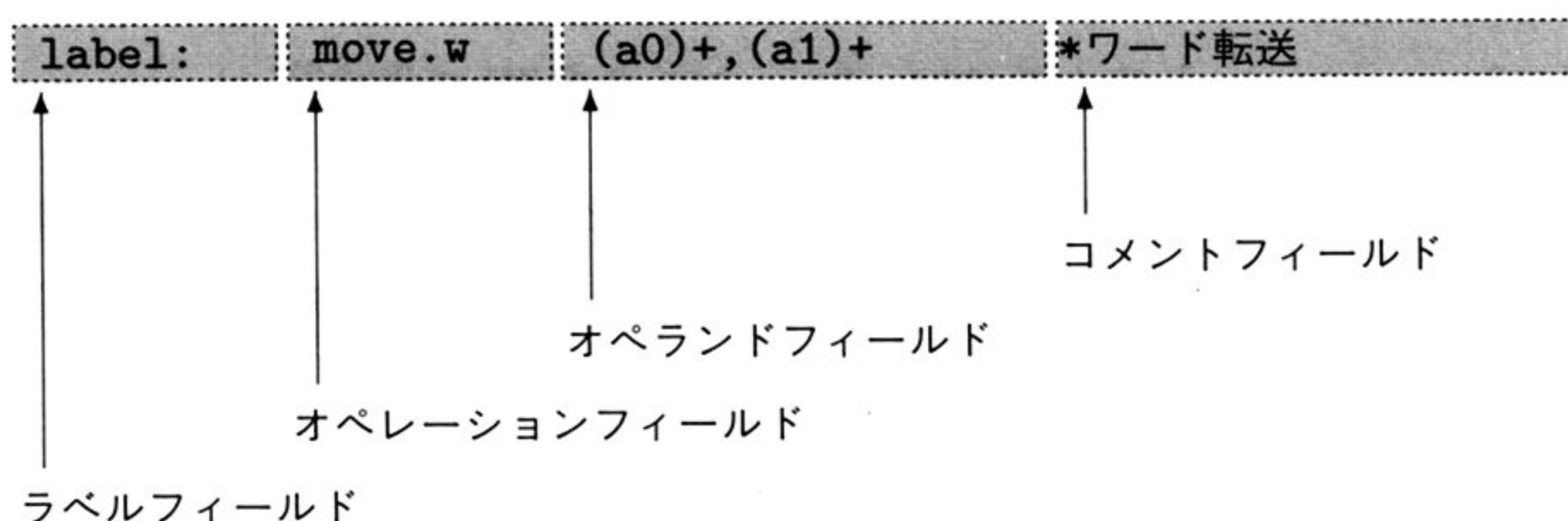



Fig. 3-1● フィールド

◆ オペレーションフィールド

オペレーションフィールドは，ラベルフィールドの後ろに 1 文字以上のスペースまたはタブコードをおいて始めます。ただしラベルフィールドが“:”で終わる場合は，オペレーションフィールドとの区切りが明確なので，スペースなどは必要ありません。また，オペレーションフィールドの内容がアセンブラ疑似命令の場合は，慣習として“.”でオペレーションフィールドを開始することが多いのですが，このピリオドは省略することもできます。

オペレーションフィールドには，アセンブリ言語命令のニーモニックを記述します。命令は，次の 3 種類に分類されます。

❖ 実行命令

68000 CPU の機械語命令に直接対応する命令です。実行命令は，アセンブラによって，2 ～ 10 バイトの機械語命令に変換されます。

実行命令の記述には，68000 CPU の標準ニーモニックを使用します。

❖ マクロ命令

複数の命令の記述をまとめて 1 つのシンボルに割り当てたもので，`.macro` 疑似命令[2]によって定義されます。この命令が使用されると，定義されている内容がその命令の位置に展開されます[3]。

マクロ命令については第 3.4 節 (P.93) で解説します。

❖ アセンブラ疑似命令

アセンブラを制御するための，アセンブラ自身に対する命令です。データ定義に関する疑似命令[4]を除いて，一般に機械語には変換されません。

アセンブラ疑似命令については第 3.5 節 (P.97) で解説します。

2)「マクロ定義の開始」を参照 (第 3.5.5 節 P.110)。

3)マクロの展開といいます。

4).dc 疑似命令 (第 3.5.6 節 P.114) などです。

実行命令の大部分と一部の疑似命令[5]では，処理するデータのサイズを指定する必要があります。これは，ニーモニックの後ろに“.”に続いてデータサイズコードを表記することで指定できます。データサイズコードには，以下の 4 種類があります。

5).dc 疑似命令などです。

- “.b”　バイトサイズの指定
- “.w”　ワードサイズの指定
- “.l”　ロングワードサイズの指定

6)意味的には .b と同じですが，これらはブランチ命令の場合にのみ慣用的に使用されます。また X680x0 HAS では，浮動小数点コプロセッサ命令などで使用された場合に単精度実数サイズ指定の意味をもちます。

- ".s"　　ショートサイズ (BRA，Bcc，BSR 命令用) の指定[6)]

データサイズが決まっている命令の場合，データサイズコードの指定を省略することができます。また，データサイズが決まっていない命令にもかかわらずデータサイズコードを指定しないと，デフォルトとして，

ワードサイズが指定された

ものとして処理されます。

List 3-1 の 3 行目ではデータサイズコードが指定されていないので，デフォルトとしてワードサイズが選択されます。したがって，2 行目と 3 行目は同じ処理を行います。

List 3-2 の場合は LEA 命令のデータサイズがロングワードサイズと決まっているので，指定を省略することができます。

List 3-1● オペレーションデータサイズ

```
1:        move.b  d0,d1     * バイトサイズのデータ転送
2:        move.w  d0,d1     * ワードサイズのデータ転送
3:        move    d0,d1     * (ワードサイズのデータ転送)
4:        move.l  d0,d1     * ロングワードサイズのデータ転送
```

List 3-2● LEA 命令

```
1:        lea.l   adr,a0    * アドレスの転送 (ロングワードサイズ)
2:        lea     adr,a0    * アドレスの転送 (ロングワードサイズ)
```

◆ オペランドフィールド

オペランドフィールドは，オペレーションフィールドの命令が必要とするオペランドを記述します。オペレーションフィールドの命令の種類によっては，オペランドが必要なかったり，複数のオペランドを必要としたりする場合があります。

複数のオペランドを記述する場合には，各オペランドの間を "," で区切って記述します。

HAS 拡張 オペランドフィールドでは，オペランドを見やすくするために "," の前後にかぎって，スペースまたはタブコードを挿入することができます。

◆ コメントフィールド

コメントフィールドは，処理の説明などを注釈として書くことで，プログラムを読みやすくするために使用します。コメントフィールドは "*" によって開始し，行末までがすべてコメント文となります。コメント文はアセンブラによってすべて無視されるので，生成される機械語には影響しません。

HAS拡張 “;”によってコメントフィールドを開始することもできます。この場合も，コメントフィールドの機能はまったく同じです。

3.2.2 識別名

識別名には，ユーザが定義する「シンボル」とアセンブラによって意味がすでに定まっている「キーワード」の2つがあります。

◆ シンボル

シンボルは，ユーザがプログラム中で自由にアドレスや定数，マクロなどを定義するための識別名です。シンボルに使用することのできる文字は，以下のとおりです。

- A ～ Z
- a ～ z
- 0 ～ 9
- ?
- @
- _ [7]
- ~ [8]
- **すべての2バイト文字(漢字など)**

7)アンダーバー(キャラクタコード $5F)です。

8)チルダ(キャラクタコード $7E)です。

このうち“0 ～ 9”と“@”を，シンボルの1文字目として使用したり，アセンブラの予約語であるレジスタ名と同名のシンボルを使用したりすることはできません。またシンボルは，一部の例外を除いて英文字の大文字／小文字を区別します。

シンボルは「数値シンボル」,「レジスタリストシンボル」,「マクロシンボル」の3種類に大別することができます。

❖ 数値シンボル

数値やアドレスを定義するための，最も一般的なシンボルです。このシンボルは，外部参照や外部定義によって他のモジュールとの間で相互に参照することが可能です。

数値シンボルはさらに，命令のロケーションアドレスを保持するためのラベルや .equ 疑似命令[9] によって定義されたシンボルのように，その値を変更することのできない不変シンボルと，.set 疑似命令[10] によって定義され，値をソースファイル中で変更可能な可変シンボルに分けられます。

また，数値シンボルは定義された値とともに，その値のもつ属性[11]を保持しています。属性が定数であるシンボルに対しては，アセンブラでサポートされているすべての演算が可能です。しかし，属性がアドレスであるシンボルに対しては,

- **定数を加減算する**

9)「不変シンボル値の定義」を参照(第3.5.4節 P.108)。

10)「可変シンボル値の定義」を参照(第3.5.4節 P.108)。

11)定数かアドレス(どのセクションに属するかなど)のどちらかになります。

● **アドレスを減算する**

という2種類の演算しかできません[12)]。

12)このことは，アドレス属性のシンボルをC言語のポインタ変数に置き換えて考えれば，容易に理解できると思います。

List 3-3● 数値シンボル

```
1:  LABEL:                          * アドレス
2:  VALUE   equ     10              * 定数
3:  LABEL2  equ     LABEL           * アドレスになる
4:  VALUE2  equ     VALUE*3         * 定数になる
5:  LABEL3  equ     LABEL2+VALUE    * 演算は可能で，結果はアドレスになる
6:  LABEL4  equ     LABEL3-LABEL    * 結果は定数になる
7:  LABEL5  equ     LABEL*10        * 演算は定義されていない (エラー)
```

❖ レジスタリストシンボル

.reg 疑似命令[13)] によってレジスタリストを定義したシンボルです。MOVEM 命令のオペランドとして使用されます。

List 3-4 ではレジスタリストシンボル reglist に，“d0-d7/a0-a6” というレジスタリストを定義しています。

13)「レジスタリストの定義」を参照 (第3.5.4節 P.109)。

List 3-4● レジスタリストシンボル

```
1:  reglist reg     d0-d7/a0-a6
2:          ...
3:          ...
4:          move.l  reglist,-(sp)   * move.l d0-d7/a0-a6,-(sp) と同じ
```

❖ マクロシンボル (マクロ命令)

.macro 疑似命令[14)] によって定義されるシンボルです。.macro 疑似命令から .endm 疑似命令[15)] までの命令群を1つのシンボルとして定義します。一度定義されると，ステートメントのオペレーションフィールドに命令として記述することによって，定義された内容が展開されます。

マクロシンボルは，他のシンボルと違って，

大文字/小文字を区別しません

ので，マクロの記述には注意してください。

List 3-5 では，マクロシンボル “test” を定義しています。“test” でも “TEST” でも，同じマクロ命令を指すことになります。

14)「マクロ定義の開始」を参照 (第3.5.5節 P.110)。

15)「マクロ定義の終了」を参照 (第3.5.5節 P.111)。

List 3-5● マクロ定義

```
1:  test    .macro
2:          .dc.b   "Test string",0
3:          .endm
```

またアセンブラによって，以下の特殊なシンボルが定義されています。

❖ “*”

現在のロケーションアドレスを保持しているシンボルのことです。たとえば，“bra *” は現在のロケーションアドレスに分岐するので，無限ループになります。

❖ ローカルラベル HAS拡張

プログラム中で一時的に分岐先などを与えるための，局所的な特殊シンボルであり，以下の3種類があります。また，ローカルラベルは，Fig. 3-2のような方法で参照します。

- "@@:"
  一般のラベルと同じように命令のロケーションアドレスを保持するが，ソースファイル中に何度もおくことができる[16]
- "@F"，"@f"
  ローカルラベル "@@:" を参照するために使用する特殊なシンボル。"@F" の書かれた位置より前方 (アドレスの増加する方向) の，最も近いローカルラベルを参照する
- "@B"，"@b"
  "@F" と同様にローカルラベルを参照するための特殊なシンボル。"@B" の書かれた位置より後方 (アドレスの減少する方向) の，最も近いローカルラベルを参照する

16)一般のラベルでは，シンボルの二重定義としてエラーになります。

```
1: @@:
2:         bra @F
3:         bra @B
4: @@:
```

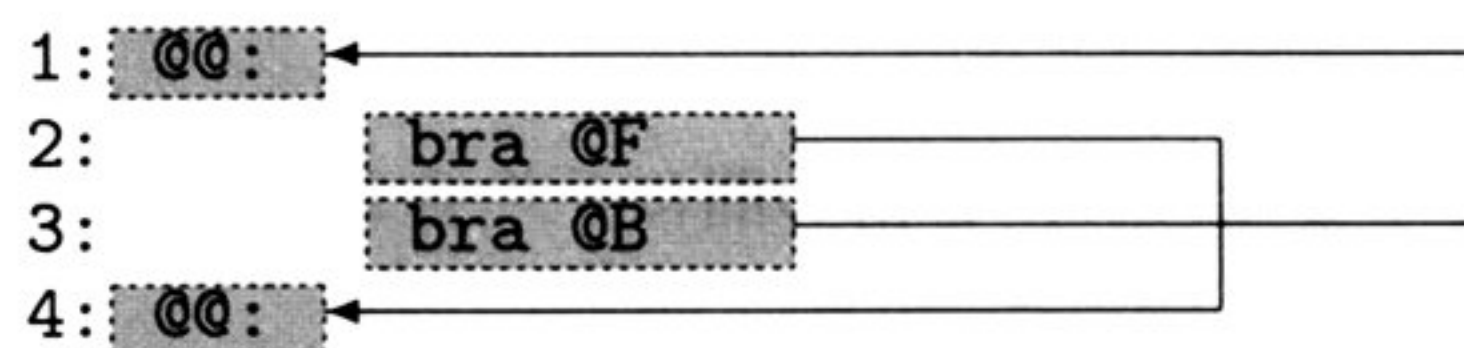



Fig. 3-2● ローカルラベル参照方法

◆ キーワード

キーワードは，アセンブラによって意味が定められている予約語です。キーワードの大文字・小文字は区別されません。キーワードには以下のものがあります。

❖ ニーモニック

各実行命令，アセンブラ疑似命令をニーモニックで表したものです。

❖ レジスタ名

アセンブリ言語の実行命令で使用するレジスタの名前[17]のことです (Table 3-1)。

17)これは68000 CPUのレジスタ名です。現在ではサポートするCPUが増えたため，Table 3-1にあるレジスタ名に加えて，それぞれのCPUで使用できるレジスタをレジスタ名として指定できます。

Table 3-1● アセンブリ言語のレジスタ名

レジスタ名の表現	レジスタの種類
D0～D7，R0～R7	データレジスタ
A0～A7，R8～R15	アドレスレジスタ
USP	ユーザスタックポインタ
SR	ステータスレジスタ
CCR	コンディションコードレジスタ
PC	プログラムカウンタ
A7，SP	スタックポインタ

3.2.3 定数

定数には,「数値定数」と「文字定数」の 2 種類があります。

◆ 数値定数

数値定数には,「10 進数」,「16 進数」,「8 進数」,「2 進数」の 4 種類があり, 32 ビットまでの数値を扱います。

- **10 進数**
  0 ～ 9 までの数字で表す。数値定数の先頭に, 特に指定がなければ, 数値は 10 進数として扱われる
- **16 進数**
  0 ～ 9, A ～ F (a ～ f) の英数字で表し, その先頭に "$" をつけて 16 進数であることを示す
- **8 進数**
  0 ～ 7 までの数字で表し, その先頭に "@" をつけて 8 進数であることを示す
- **2 進数**
  0 または 1 の数字で表し, その先頭に "%" をつけて 2 進数であることを示す

いずれの定数も値を読みやすくするために, 各桁の間に "_" をおくことができます。たとえば, 次の例のように。

例 :　`12345　@277　%1010_1111_0101_0000　$ffd018`

◆ 文字定数

文字定数は文字列を " ' " または " " " によって囲むことで表します。文字列の各文字は ASCII コードに変換されて扱われます。

List 3-6● 文字定数

```
1:        .dc.l   'ab'    * ロングワードサイズ $00006162 をメモリに確保
2:        .dc.b   "ABC"   * バイトサイズ $41, $42, $43 をメモリに確保
```

3.2.4 演算子

演算子には「単項演算子」と「2 項演算子」の 2 種類があります。

そして, 演算子には次のような優先順位があり, 数値の小さいものほど優先順位が高くなります。

1. **( ) で囲まれた式**
2. **単項演算子** (+, -, .not., .high., .low., .highw., .loww.)
3. **乗除算, 剰余, シフト** (*, /, .mod., .shr., .shl., .asr.)
4. **加減算** (+, -)
5. **大小比較 0** (.eq., .ne., .lt., .le., .gt., .ge.)

6. **大小比較 1** (.slt., .sle., .sgt., .sge.)
7. **論理 AND** (.and.)
8. **論理 OR，排他的 OR** (.or., .xor.)

◆ 単項演算子

単項演算子は 1 つのオペランドのみを取り，それに対して演算を行う演算子で，次の 7 種類が含まれます。

- “+”　値が正であることを示す
- “−”　値が負であることを示す (符号を反転)
- “.not.”　値の各ビットを反転 (論理 NOT)
- “.high.”　値の下位ワードの上位バイトを得る
- “.low.”　値の下位ワードの下位バイトを得る
- “.highw.”　値の上位ワードを得る
- “.loww.”　値の下位ワードを得る

◆ 2 項演算子

2 項演算子は「算術演算子」と「比較演算子」の 2 種類に分類できます。どちらも 2 つのオペランドに対して演算を行う演算子です。

❖ 算術演算子

2 つのオペランドの間で四則演算などを行う算術演算子です。

- “+”　加算
- “−”　減算
- “*”　乗算
- “/”　除算
- “.mod.”　剰余
- “.shr.”, “>>”　指定ビット数だけ右へ論理シフトする
- “.shl.”, “<<”　指定ビット数だけ左へ論理シフトする
- “.asr.”　指定ビット数だけ右へ算術シフトする。つまり最上位ビットを符号とみなし，その内容を変化させない
- “.and.”　2 数の論理積を得る (論理 AND)
- “.or.”　2 数の論理和を得る (論理 OR)
- “.xor.”　2 数の排他的論理和を得る (排他的 OR)

❖ 比較演算子

2 つのオペランドを比較する比較演算子です。比較結果が真のときは “−1”，偽のときは “0” を式の値とします。

- “.eq.”, “=”　2 数が等しいならば真
- “.ne.”, “<>”　2 数が等しくないならば真

次の大小比較は，符号をもたない数として行われます。

- “.lt.”，“<”　左オペランドが右オペランドより小さいならば真
- “.le.”，“<=”　左オペランドが右オペランドより小さいか等しいならば真
- “.gt.”，“>”　左オペランドが右オペランドより大きいならば真
- “.ge.”，“>=”　左オペランドが右オペランドより大きいか等しいならば真

次の大小比較は，符号つき数として行われます。

- “.slt.”　左オペランドが右オペランドより小さいならば真
- “.sle.”　左オペランドが右オペランドより小さいか等しいならば真
- “.sgt.”　左オペランドが右オペランドより大きいならば真
- “.sge.”　左オペランドが右オペランドより大きいか等しいならば真

3.3 ……………… セクションとモジュール化機能

アセンブリ言語では，プログラムを機械語コードとしての目的別にいくつかのまとまりに分割できるようになっています。

またアセンブリ言語では，プログラムを機能ごとにモジュール化して別々のソースファイルに分割し，おのおのをアセンブルしたオブジェクトファイルを最後にリンカによって結合するような，モジュール別開発ができるようになっています。

このセクションではプログラム作成における，このようなアセンブリ言語の仕様について説明します。

3.3.1 セクション

アセンブリ言語によって作成されるプログラムは，最終的には機械語コードとしてメモリ上のあるアドレスに配置されて実行されます。そのメモリは，プログラムの命令コード自身やプログラムによって使用されるデータ，そしてスタックエリアなどのいろいろな目的に使用されます。アセンブラでは，アセンブリ言語の段階からこうした目的別に，プログラムをいくつかのまとまりに分割して作成できるようになっています。そして，このまとまりのことをセクションと呼びます。

プログラムを複数のモジュールに分割して作成した場合も，命令コードやデータ領域などをおのおの同じセクションに指定しておけば，リンカでセクションごとにプログラムが結合され，プログラム全体としてのセクション分けがなされるようになります。

アセンブラで通常使用できるセクションには，次の 4 種類があります。

❖ **テキストセクション**

.text 疑似命令[1] によって指定されます。一般的に，実行命令のコード自身や，プログラム中で変更されることのない固定データをおくために使用されます。

❖ **データセクション**

.data 疑似命令[2] によって指定されます。一般的に，初期値が定まっている変更可能なデータをおくために使用されます。

1)「テキストセクションの宣言」を参照 (第 3.5.2 節 P.101)。

2)「データセクションの宣言」を参照 (第 3.5.2 節 P.101)。

❖ **ブロックストレージセクション**

.bss 疑似命令[3] によって指定され，初期値の定まっていない変更可能なデータをおくために使用されます。このセクションと次のスタックセクションでは，実行プログラムにセクションの領域の大きさ以外の情報は残りません。したがってこのセクションにおくことのできるのは，領域を確保する .ds 疑似命令[4] のみで，実行命令や .dc 疑似命令[5] などをおくことはできません。

❖ **スタックセクション**

.stack 疑似命令[6] によって指定され，プログラムの使用するスタック領域をおくために使用されます。このセクションもブロックストレージセクション同様，領域を確保することしかできません。

3)「ブロックストレージセクションの宣言」を参照(第3.5.2節 P.102)。

4)「メモリ領域の確保」を参照(第3.5.6節 P.115)。

5)「定数データの定義」を参照(第3.5.6節 P.114)。

6)「スタックセクションの宣言」を参照（第3.5.2節 P.102)。

各セクションは，それぞれ独立したロケーションカウンタをもっています。ロケーションカウンタは0で初期化されていて，実行命令やデータの定義，領域の確保などによって増加します。たとえ他のセクションへ切り替えられても，それぞれの値は保持されているので，再び切り替えられたときには，以前の値から続けることができます。

また，ロケーションカウンタがもっているのは各セクションの先頭からの相対アドレスであり，プログラムが実行されるときは，そのプログラムが配置されたメモリ上の絶対アドレスに置き換えられます。

3.3.2　外部参照

複数のソースファイルによってモジュール別開発を行う場合，あるソースファイルから他のソースファイル中のシンボルを参照する必要が生じます。このように，他のソースファイルからシンボルを参照することを**外部参照** といいます。また逆に，シンボルを他のソースファイルから参照できるようにすることを**外部定義** といいます。

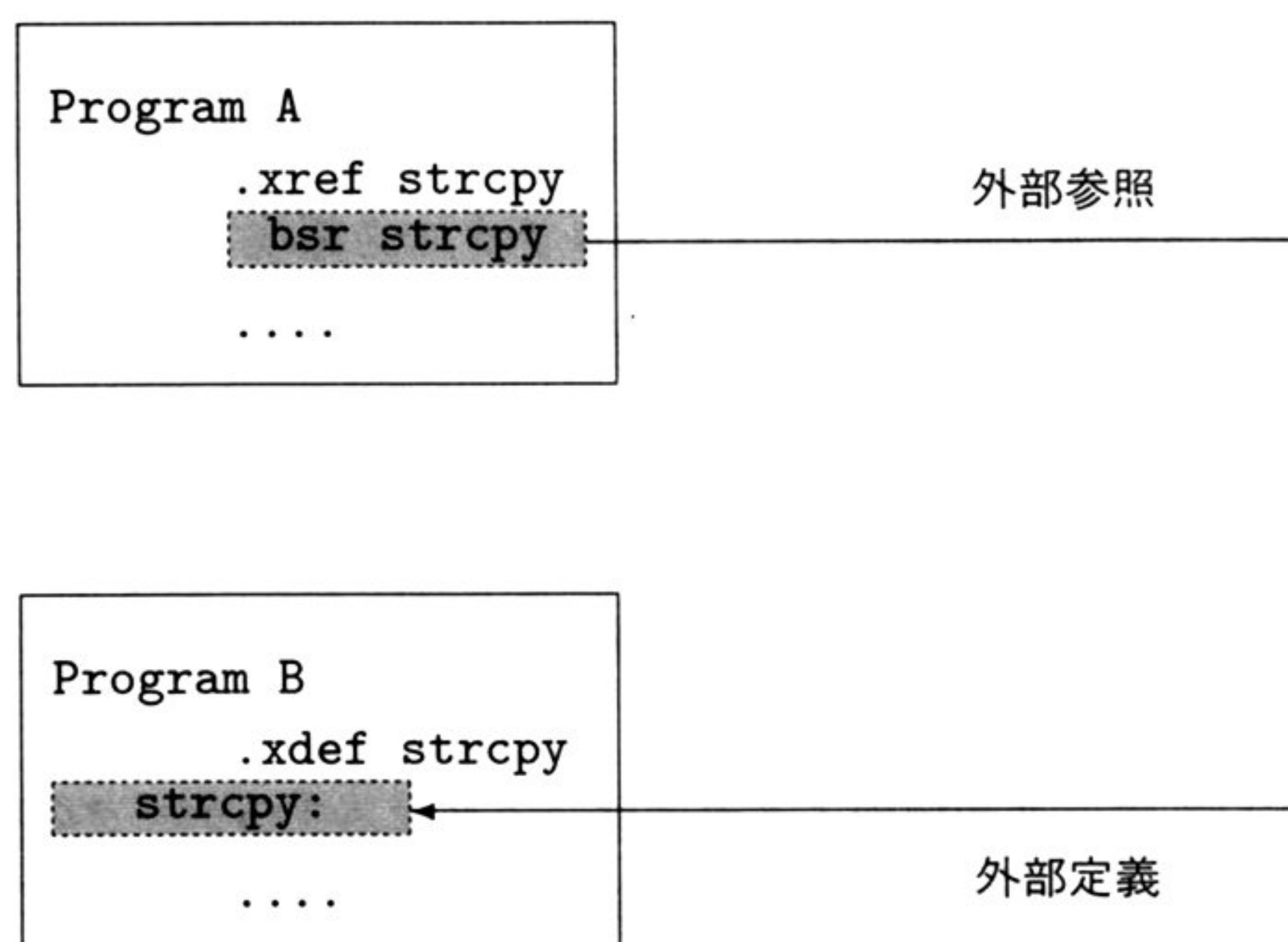
通常シンボルを参照する際に，そのシンボルがソースファイル中に存在しなければ，アセンブラはシンボルが定義されていないとしてエラーを発生します。ですから，外部参照を行うときには，そのシンボルが外部参照シンボルであることを宣言しておく必要があります。また，一般的にシンボルの定義が有効なのはそのソースファイルの中だけなので，外部からの参照を可能にするためには，そのシンボルが外部定義シンボルであることを宣言する必要があります。これらの宣言を行うためには，外部参照では **.xref** 疑似命令[7]，外部定義では **.xdef** 疑似命令[8]を使用します (Fig. 3-3)。これらの疑似命令によって宣言を行うことで，アセンブラはオブジェクトファイルにこうした外部シンボルの情報を出力するようになります。

7)「外部参照名の宣言」を参照(第3.5.3節 P.107)。

8)「外部定義名の宣言」を参照(第3.5.3節 P.107)。

こうして作成された複数のオブジェクトファイルは，リンカによって1つの実行可能ファイルに結合されます。リンカは，オブジェクトファイルの中の外部参照に関する情報に対して，他のオブジェクトファイル中の対応する外部定義の情報を用いてアドレスの決定を行います。

```
Program A
        .xref strcpy
        bsr strcpy
        ....

Program B
        .xdef strcpy
strcpy:
        ....
```

外部参照

外部定義

Fig. 3-3● 外部定義と外部参照

3.3.3 共通データエリア(コモンエリア)

複数のプログラムから共通に使用されるデータ領域は，ある 1 つのソースファイル中でその定義を行い，他のソースファイルはそれを外部参照することによって使用することができます。しかし共通データエリア (コモンエリア) は，すべてのソースファイルで.comm 疑似命令によって[9]同じ宣言をしておくことで使用できます。

つまり，共通データエリアはデータ領域の定義を行う必要がなく，リンクの際にリンカが各プログラム中の宣言をもとに，1 つの領域をブロックストレージセクションの一部として割り当てます。

9)「コモンエリアの指定」を参照 (第 3.5.2 節 P.103)。

3.3.4 相対セクション HAS 拡張

X68000 の標準ウィンドウ環境である SX–Window では，メモリ効率を向上させるなどの目的から，リエントラント なプログラム[10]を実行させることができます。**HAS** には，こうしたリエントラントなプログラム作成を支援するために，新たに相対セクション というセクションが使用できるようになっています。

10)SX–Window では，OBJR 型モジュールと呼ばれます。

11)スタックも含みます。

◆ リエントラントなプログラム

一般的に，プログラムは大きく「命令コード」と「データ[11]」に分けることができますが，命令コードの部分は原則として，プログラムの実行を通じてその内容が変更されることはありません。そこで命令コードや固定のデータ領域など，内容の変更されない領域と，内容の変更されるデータ領域を完全に分離してしまうことを考えます。すると，同じプログラムを複数同時に実行させるような場合，各タスクごとに確保する必要があるのはデータ領域のみになり，命令コードの領

域を完全に共有することができるようになるわけです。このようなプログラムをリエントラント[12] なプログラムといいます。

12)再入可能ともいいます。

Human68k のようなシングルタスクの環境下では，このような工夫はそれほど意味のないことですが，SX–Window のようなマルチタスク環境[13]においては，メモリ効率の面からむしろ積極的にリエントラントなプログラム作成を行うことが望まれます。

13)厳密にはSX–Windowの場合，プログラム自体に他のタスクに制御を渡すことを意識したコーディングが必要となるので，疑似的なマルチタスクといえます。

◆ リエントラントプログラムの実現

それでは，実際にリエントラントなプログラムを実現するためには，どのような手法が必要となるでしょうか？

通常のプログラムにおいてデータ領域のアクセスは，アクセスするデータのアドレスを直接指定することによって行われます。ところが，このようなプログラムの命令コードをタスク間で共有すると，おのおののタスクが同じデータ領域をアクセスしてしまいます。

そこで，データ領域のアドレスをレジスタに保持するようにし，すべてのデータアクセスは，そのレジスタの値からの相対値（オフセット値）で行うようにします。各タスクはそれぞれ固有のレジスタ値をもっていますから，タスクの実行開始時におのおののデータ領域のアドレスをレジスタで渡すように決めておけば，データ領域を完全に分離することが可能になります。

List 3-7 は，このようなデータアクセスをアセンブリ言語で実現した例です。`.offset` 疑似命令[14]によって，シンボルにデータ領域のアドレスからのオフセット値を与えています。

14)「オフセットテーブルの定義開始」を参照（第3.5.2節 P.102）。

List 3-7● リエントラントプログラムの実現

```
1:          .offset 0
2:  x:      .ds.l   1
3:  y:      .ds.l   1
4:  var:    .ds.w   1
5:          ...
6:          ...
7:  * a5 レジスタには，データ領域のアドレスが
8:  * 格納されているものとする
9:          move.w  var(a5),d0
```

ところが，複数のソースファイルによってモジュール別開発を行う場合には，この書き方では問題があります。なぜならばモジュール別開発の場合，各モジュールがどのくらいのデータ領域を必要とするのかはリンクするまでわからないからです。そのためそれぞれのソースファイルが，別々にアドレスのオフセット値を求めることができなくなります。

したがって実際には，プログラム中のすべてのデータのオフセット値を1つのファイル中で定義して，それを各ソースファイルからインクルードするという方法がとられます。しかしこの方法を用いると，モジュールの局所性は失われてしまいます。

相対セクションは，こうした問題点を解決するために **HAS** が拡張した疑似的なセクションです。この相対セクションを用いた例を List 3-8 に示します。

List 3-8 ● 相対セクションの使用例

```
1:          .rbss
2:  x:      .ds.l   1
3:  y:      .ds.l   1
4:  var:    .ds.w   1
5:          ...
6:          ...
7:  * a5 レジスタには，データ領域のアドレスが
8:  * 格納されているものとします
9:          move.w  var(a5),d0
```

ご覧のとおり List 3-7 の .offset が .rbss に変わっただけですが，List 3-7 ではアセンブルのときに .offset 疑似命令によって各シンボルにオフセット値が定義されてしまうのに対して，List 3-8 ではオフセット値はリンクの際にリンカによって決定されます[15]。したがって，前述したようなモジュール別開発による問題も解消できます。

15)このとき，リンカは 16 ビットのオフセット値を最大限利用できるようにするため，オフセット値を -$8000 から開始します。

◆ 相対セクション疑似命令

データをおくためのセクションに .data, .bss, .stack の各セクションが存在するように，相対セクションにも対応する 3 種類のセクションがあります。

さらに相対セクションは，そのアクセス方法によって 2 種類に分類されます。

1. セクション領域の大きさが 64K バイト以下の場合

データは，ディスプレイスメントつきアドレスレジスタ間接形式によってアクセスすることが可能です。実際には，各セクション上のアドレスを用いて List 3-9 のようにアクセスされます。

List 3-9 ● 64K バイトを超えないアクセス

```
1:  * a5 レジスタには，データ領域のアドレスが
2:  * 格納されているものとする
3:          move.w  var(a5),d0
```

2. セクション領域の大きさが 64K バイトを超える場合

この場合，データ領域をポイントするレジスタからの相対値が 64K バイトを超えるため，68000 CPU の制約から，ディスプレイスメントつきアドレスレジスタ間接形式を用いた 1 命令でのアクセスができません。そこで，List 3-10 のように 2 つの命令によってデータへのアクセスを行います[16]。

16)その分，速度は遅くなります。

List 3-10 ● 64K バイトを超えるアクセス

```
1:          move.l  #var,d7
2:          move.w  (a5,d7.l),d0
```

つまり合計で，次のように 6 種類の相対セクションが存在することになります。

- **64K バイト以内の領域** ：.rdata, .rbss, .rstack
- **64K バイトを超える領域** ：.rldata, .rlbss, .rlstack

また共通データエリア(コモンエリア)に対しても,

- **64Kバイト以内の領域**:　.rcomm
- **64Kバイトを超える領域**:　.rlcomm

という2種類の対応する相対共通データエリアが存在します。

これらの相対セクション疑似命令を使用する場合には,アセンブル時にコマンドラインで“-r”スイッチ[17]を指定する必要があります。また,相対セクションのデータ領域を指すレジスタの初期化や初期値つきデータセクション[18]の内容の初期化など,相対セクションを使用するにあたっての初期化は自動的に行われるわけではありません[19]。こうしたことは,

プログラマがプログラム上で責任をもって行う必要がある

ので注意が必要です。

17)相対セクション命令の使用許可です。

18).rdataと.rldataです。

19)同じくリエントラントなプログラムを実行するOS/9では,OSがめんどうをみてくれます。

3.4 ……………………………………マクロ機能

マクロとは，複数の命令の記述をまとめて 1 つのシンボルに割り当てたものです。このようにして割り当てられたシンボルはマクロ命令と呼ばれ，以後，実行命令や疑似命令のような命令の一種として使うことができるようになります。

このセクションでは，マクロ機能の詳細について説明します。

3.4.1 マクロ命令の定義

マクロは，.macro 疑似命令[1]を用いて次のように定義します。

```
<シンボル>   .macro [<仮引数>[,<仮引数> … ] ]
             命令の記述（マクロ定義ブロック）
             ...
             ...
             .endm
```

こうして定義されたマクロ命令は，次のように使用します。

```
             <シンボル> [<実引数>[,<実引数> … ] ]
```

マクロ命令が使用されると，マクロ命令の位置に定義されている内容を展開してアセンブルが行われます。その際，マクロ定義のときに使用された <仮引数> は，<実引数> と置き換えられて展開されます[2]。

3.4.2 特殊マクロ　HAS 拡張

マクロ命令の特殊な形式として，繰り返し疑似命令があります。繰り返し疑似命令には，“.rept[3]”，“.irp[4]”，“.irpc[5]” があります。

- .rept 疑似命令
  .endm 疑似命令[6]までの命令の記述を指定した回数だけ繰り返す
- .irp 疑似命令
  .endm 疑似命令までの命令の記述を，<仮引数> に <実引数> の内容を割り当てながら繰り返す

1)「マクロ定義の開始」を参照（第3.5.5節 P.110）。

2) マクロ定義ブロック中で，他のマクロを展開することができます。このネスティングは 32 重まで可能です。

3)「繰り返しの開始」を参照（第 3.5.5 節 P.112）。

4)「不定回数の繰り返しの開始」を参照（第 3.5.5 節 P.113）。

5)「不定回数の文字繰り返しの開始」を参照（第 3.5.5 節 P.113）。

6)「マクロ定義の終了」を参照（第 3.5.5 節 P.111）。

- `.irpc` 疑似命令
  `.endm` 疑似命令までの命令の記述を，<仮引数> に指定した <文字列> の文字を 1 文字ずつ割り当てながら繰り返す

特殊マクロは通常のマクロと違い，命令の記述を割り当てるシンボルはありません。マクロ展開は，マクロの定義が行われた位置で行われます。

3.4.3　マクロ内疑似命令

7)「局所的シンボルの定義」を参照 (第 3.5.5 節 P.111)。

8)「マクロ展開の打ち切り」を参照 (第 3.5.5 節 P.111)。

9)「条件付きアセンブル」を参照 (第 3.5.7 節 P.117)。

マクロ定義ブロックの中だけで使用できる疑似命令に，“`.local`[7)]”，“`.exitm`[8)]” があります。

- `.local` 疑似命令
  マクロ定義ブロック内でのみ有効な局所的シンボルを定義する。マクロ定義ブロック内でラベルの定義を行うと，そのマクロが 2 回以上展開された際にシンボルの二重定義でエラーが発生してしまう。しかし，`.local` 疑似命令によって定義された局所的シンボルは，他のシンボル定義と重ならないような名前に変換される
- `.exitm` 疑似命令
  マクロ展開を途中で打ち切る。`.if` 疑似命令[9)]などによる条件アセンブルで，ある条件によってその後のマクロ展開が不要になる場合などに使用する

3.4.4　マクロ演算子　HAS 拡張

マクロ機能を使用する際に，引数の展開を制御するなどの目的で特殊な演算子 (マクロ演算子) を使用することができます。

- “&”
  <仮引数> から <実引数> への置き換えを，単語の途中や文字列の中においても行う。通常，<仮引数> から <実引数> への置き換えは語単位で行われるため，<仮引数> が単語の途中や文字列の中にある場合には置き換えは行われないが，“&” を <仮引数> の先頭におくことによって，これらの場合にも置き換えができるようになる
  また，マクロ定義ブロック中で “&” を記述する必要がある場合は，代わりに “&&” と記述する
  たとえば，List 3-11 の 2，3 行目の “chr” は <仮引数> とみなされるが，“&” がないと，2 行目は文字列，3 行目は単語の途中として <仮引数> として扱われなくなる

List 3-11 ● & 演算子

```
1:  cmpbra  .macro  chr
2:          cmpi.b  #"&chr",d0
3:          beq     table_&chr
4:          .endm
```

- “!”

<実引数> を記述するときに，“!” の後ろの 1 文字の特殊な機能を失わせる。たとえば実引数中では，“,” は各引数の区切り，スペースやタブは引数の終了という特殊な意味をもっているが，“!” をこれらの文字の前におくことで，これらは単なる引数の文字として扱われるようになる

次のマクロ test に，List 3-13 のような引数を与えて展開することを考えよう

List 3-12 ● マクロ定義

```
1:  test    .macro  msg
2:          .dc.b   "&msg",0
3:          .endm
```

List 3-13 ● マクロ呼び出し (その 1)

```
1:          test    Hello,world
```

ところが，引数中の “,” が区切りと解釈されるため，仮引数 msg には，“Hello” という文字列だけが渡されてしまう。これを意図どおり展開するためには，List 3-14 のように “!” を使用する

List 3-14 ● マクロ呼び出し (その 2)

```
1:          test    Hello!,world
```

- “<” ～ “>”

<実引数> を記述するときに，この文字によって囲まれた文字列をすべて文字どおりに扱う

“!” と同様に，特別な意味をもった文字の機能を失わせるが，この演算子の場合，“<” から “>” によって囲まれた文字列すべてに有効となる。List 3-12 のマクロ test の展開は，List 3-15 のように記述することもできる

List 3-15 ● マクロ呼び出し (その 3)

```
1:  test    <Hello,world>
```

- “%”

この文字の後ろにあるシンボルを文字列としてではなく，そのシンボルに定義されている値を取り出して扱う。“%” の後ろにシンボルがない場合や，そのシンボルに定義されている値が定数でない場合は，“%” は通常の文字とし

て扱われる

List 3-16 のマクロ **nlabel** は，シンボル **sym** の値によって List 3-17，List 3-18 のように展開される

List 3-16● マクロ定義

```
1:  nlabel  .macro
2:  LABEL_%sym:
3:          .endm
```

List 3-17● sym=0 のとき

```
1:  LABEL_0:
```

List 3-18● sym=10 のとき

```
1:  LABEL_10:
```

3.5 アセンブラ疑似命令

アセンブラ疑似命令は，アセンブル動作を制御したり，作業に関する情報をアセンブラに対して与えたりする命令です。この命令は，一部を除いて一般にオブジェクトコードには変換されません。

アセンブラ疑似命令は，機能によって次のように分類されます。

- **アセンブラ制御**
- **セクション指定**
- **外部名の宣言**
- **シンボルの定義**
- **マクロ制御**
- **データの定義**
- **条件つきアセンブル**
- **リストファイル制御**
- **シンボリックデバッグ情報の指定**

3.5.1 アセンブラ制御

ソースコードの挿入やアセンブルの終了指示など，アセンブル動作に対しての一般的な制御を行います。この疑似命令には以下の 7 種類があります。

.include	**ソースコードの挿入**
.request	**リンク時のライブラリ指定**
.end	**プログラムの終了指定**
.org	**ロケーションカウンタの指定**
.comment	**コメント行の指定**
.fail	**エラーの生成**
.cpu	**アセンブル対象命令セットの指定**

.include　ソースコードの挿入

書式：　.include <ファイル名>

機能：　ソースファイル中に他のファイルを挿入します。

解説：　ソースファイル中の .include 疑似命令のある位置に，<ファイル名> で指定するインクルードファイルの内容を挿入します。

インクルードファイルの中でさらに他のファイルをインクルードすることもできます[1]。

List 3-19 では，インクルードファイル IOCSCALL.MAC の内容がソースファイルの中に取り込まれます。

1)最大８重までネスト可能です。

List 3-19 ● .include 疑似命令

```
1:          .include IOCSCALL.MAC
```

.request　リンク時のライブラリ指定

書式：　.request <ファイル名> [,<ファイル名> …]

機能：　リンカによってリンクされる際に使用するライブラリを指定します。

解説：　指定した <ファイル名> がオブジェクトファイル中に出力されます。リンカはリンクの際に，指定したファイルをライブラリとして自動的にリンクします。

List 3-20 では，ライブラリファイル名 MYLIB.L がオブジェクトファイルの中に出力されます。このオブジェクトファイルをリンクすると，自動的に MYLIB.L がライブラリファイルとしてリンクされます。

List 3-20 ● .request 疑似命令

```
1:          .request MYLIB.L
```

.end　プログラムの終了指定

書式：　.end [<ラベル>]

機能：　ソースプログラムの終了を指定します。

解説：　アセンブラに対して，ソースプログラムの終了を通知します。この疑似命令の後にあるソースコードはすべて無視されます。

<ラベル> の指定をした場合には，プログラムの実行開始アドレスをラベルのアドレスとします。<ラベル> の指定がない場合は，プログラムの実行はテキストセクションの先頭アドレス[2]から開始されます。

2)複数のオブジェクトファイルをリンクする場合は，最初にリンクされるオブジェクトファイルのテキストセクションの先頭アドレスになります。

List 3-21● .end 疑似命令

```
1:  * ソースプログラムはこの行で終了する
2:  * プログラムの実行は，ラベル execaddr の
3:  * アドレスから行われる
4:          .end     execaddr
```

.org　ロケーションカウンタの指定

書式： .org <式>

機能： ロケーションカウンタの値を設定します。

解説： 現在のセクションのロケーションカウンタの値を <式> の値とします。<式> の値は定数でなければならず，前方参照値や外部参照値，アドレス値などは使えません。

List 3-22● .org 疑似命令

```
1:  * 現在のセクションのロケーションカウンタ値を
2:  * $1234 にする
3:          .org     $1234
```

.comment　コメント行の指定

書式： .comment <文字列>

機能： コメント行を開始します。

解説： .comment 疑似命令のある行から再び <文字列> の現れる行までを，すべてコメント行とします。コメントをネストすることはできません。List 3-23 では，.comment 疑似命令のある 1 行目から再び文字列 “comend” の現れる 3 行目までがすべてコメント行となります。

List 3-23● .comment 疑似命令

```
1:          .comment comend
2:  この行はコメント行になる
3:          comend
4:          move.l  (a0)+,(a1)+
5:          ....
```

.fail　エラーの生成

書式： .fail <式>

機能： エラーを生成します。

解説： <式> の値が真 (0 以外) ならば，アセンブル時にエラーを発生させます。 <式> の値は定数でなければならず，前方参照値や外部参照値，アドレス値などは使えません。

List 3-24 ● .fail 疑似命令

```
1:  * シンボル LIMIT が $800 以上なら
2:  * エラーを発生し，アセンブルを中断する
3:          .fail   LIMIT.ge.$800
```

.cpu　アセンブル対象命令セットの指定

書式： .cpu 68000 ··· 68000 命令セットの指定
.cpu 68010 ··· 68010 命令セットの指定

機能： アセンブルの対象となる命令セットを指定します。

解説： アセンブルの対象となる命令セットを指定します。デフォルトでは 68000 の命令セットのアセンブルが可能ですが，.cpu 68010 によって 68010 で拡張された命令セットのアセンブルが可能になります。ただし，アセンブルによって生成されるオブジェクトが実行できるかどうかは実行環境に依存します。

拡張： **X680x0 HAS** ではサポートする CPU が増えたため，68000，68010 のほかに 68020，68030，68040 を指定することができます。

List 3-25 ● .cpu 疑似命令

```
1:  * 68010 命令セットを指定して，68010 用の命令である
2:  * RTD 命令をアセンブルしている
3:          .cpu    68010
4:          rtd     #-4
```

3.5.2 セクション指定

オブジェクトの領域を分割するセクションの指定を行います。この疑似命令には以下の 14 種類があります。

.text	テキストセクションの宣言
.data	データセクションの宣言
.bss	ブロックストレージセクションの宣言
.stack	スタックセクションの宣言
.offset	オフセットテーブルの定義開始
.comm	コモンエリアの指定
.rdata .rbss .rstack .rldata .rlbss .rlstack	相対セクションの宣言 HAS 拡張
.rcomm .rlcomm	相対コモンエリアの指定 HAS 拡張

.text テキストセクションの宣言

書式： .text

機能： テキストセクションの宣言を行います。

解説： プログラムのテキストセクション[3)]の開始を宣言します。

3)プログラムコードの定義部です。

List 3-26● .text 疑似命令

```
1:  * テキストセクションの宣言を行う
2:            .text
```

.data データセクションの宣言

書式： .data

機能： データセクションの宣言を行います。

解説： プログラムのデータセクション[4)]の開始を宣言します。

4)初期値つきデータ部です。

List 3-27 ● .data 疑似命令

```
1:  * データセクションの宣言を行う
2:          .data
```

.bss　ブロックストレージセクションの宣言

書式： .bss

機能： ブロックストレージセクションの宣言を行います。

解説： プログラムのブロックストレージセクション[5]の開始を宣言します。

5)初期値なしデータ部です。

List 3-28 ● .bss 疑似命令

```
1:  * ブロックストレージセクションの宣言を行う
2:          .bss
```

.stack　スタックセクションの宣言

書式： .stack

機能： スタックセクションの宣言を行います。

解説： プログラムのスタックセクションの開始を宣言します。

List 3-29 ● .stack 疑似命令

```
1:  * スタックセクションの宣言を行う
2:          .stack
```

.offset　オフセットテーブルの定義開始

書式： .offset <式>

機能： オフセットテーブルの定義を開始します。

解説： .ds 疑似命令によって構成するオフセットテーブルの定義を開始します。<式> には定義するオフセットテーブルの開始アドレスを指定しますが，この値は定数でなければならず，前方参照値や外部参照値，アドレス値などは使えません。

オフセットテーブルは一種の疑似セクションで，メモリ上に構成されるデータ構造を .ds 疑似命令で疑似的に構成することで，開始アドレスからのオフセット値を定数データで与えるものです。したがって，

オフセットテーブルで定義したデータ構造が，プログラムのその位置に実際に構成されるわけではありません。

また，オフセットテーブルは .offset 疑似命令によって開始し，他のセクション命令によってセクションが変更されるか，アセンブルが終了するまで続きます。オフセットテーブルの中には，機械語に変換されるような実行命令や疑似命令を記述することはできません。

List 3-30 のように定義します。すると，たとえば A1 レジスタにこのデータ構造の先頭アドレスが入っていると仮定すると，PREV(A1) というアドレス形式によって PREV フィールドの値を指すことができるようになります。

List 3-31 は，List 3-30 の定義とほぼ同じ意味になりますが，List 3-31 の定義では，後からデータ構造の内容に追加があった場合，追加されたフィールドの後ろのオフセット値も変更する必要があります。

List 3-30 ● .offset 疑似命令

```
1:          .offset 0
2:  NAMEPTR:.ds.l    1          * 4bytes
3:  COUNT:   .ds.w    1          * 2bytes
4:  PREV:    .ds.l    1          * 4bytes
5:  NEXT:    .ds.l    1          * 4bytes
```

List 3-31 ● List 3-30 とほぼ等価な例

```
1:  NAMEPTR equ       0
2:  COUNT   equ       4
3:  PREV    equ       6          * 4+2
4:  NEXT    equ       10         * 4+2+4
```

.comm　コモンエリアの指定

書式：　.comm　<ラベル>,<式>

機能：　コモンエリアの指定を行います。

解説：　コモンエリアのラベルとそのサイズを指定します。<式> にはコモンエリアのサイズを指定しますが，前方参照値や外部参照値，アドレス値などは使えません。

コモンエリアは，別々にアセンブルされたプログラムが共通のデータ領域として共有する領域です。リンカは，リンクの際に各オブジェクトファイルで同じラベルをもつすべてのコモンエリアを，ブロックストレージセクションの一部として同じアドレスに割り付けます。このとき，同じラベルをもつコモンエリアのサイズが異なる場合は，その中で最大のサイズが割り当てられます。

List 3-32では，シンボル comvar をコモンエリアのラベルとし，4バイトの領域を割り当てます。複数のプログラムに同じ表記があっても，リンクの際にすべてのプログラムが同じ領域を指すように処理されます。

List 3-32 ● .comm 疑似命令

```
1:          .comm   comvar,4
```

.rdata/.rbss/.rstack 相対セクションの宣言 .rldata/.rlbss/.rlstack HAS拡張

書式： `.rdata` ······相対データセクションの宣言(64Kバイト以内)
`.rldata` ·····相対データセクションの宣言(64Kバイト以上)
`.rbss` ·······相対ブロックストレージセクションの宣言(64Kバイト以内)
`.rlbss` ······相対ブロックストレージセクションの宣言(64Kバイト以上)
`.rstack` ····· 相対スタックセクションの宣言(64Kバイト以内)
`.rlstack` ···· 相対スタックセクションの宣言(64Kバイト以上)

機能： 相対セクションの宣言を行います。

解説： プログラムのデータ領域を先頭アドレスからの相対値で参照するような，相対セクションの宣言を行います。

従来は，これらの命令を使用するためには，アセンブルの際にコマンドライン上で-r スイッチ[6]をつける必要がありましたが，**X680x0 HAS** ではこのオプションスイッチは不要になっています。

6)相対セクション命令の使用許可です。

List 3-33 ● .rldata 疑似命令

```
1:  * 64Kバイト以上の相対データセクションの宣言を行う
2:          .rldata
```

.rcomm/.rlcomm 相対コモンエリアの指定 HAS拡張

書式： `.rcomm <ラベル>,<式>` ······ 相対コモンエリアの指定(64Kバイト以内)
`.rlcomm <ラベル>,<式>` ····· 相対コモンエリアの指定(64Kバイト以上)

機能： 相対コモンエリアの指定を行います。

解説： 相対コモンエリアのラベルとそのサイズを指定します。 <式> には，相対コモンエリアのサイズを指定しますが，前方参照値や外部参照値，アドレス値等は使えません。
相対コモンエリアは，コモンエリアと同様に複数のプログラムから共通のデータ領域として共有する領域ですが，その領域は先頭アドレスからの相対値で参照されます。
従来は，これらの命令を使用するためには，アセンブルの際にコマンドライン上で `-r` スイッチ[7]をつける必要がありましたが，**X680x0 HAS** ではこのオプションスイッチは不要になっています。

7)相対セクション命令の使用許可です。

List 3-34● .rcomm 疑似命令

```
1:  * シンボル comvar を 64K バイト以内の相対コモンエリアのラベルとし
2:  * 4 バイトの領域を割り当てる。複数のプログラムに同じ表記が
3:  * あっても，リンクの際にすべてのプログラムが同じ領域を指すように
4:  * 処理される
5:          .rcomm  comvar,4
```

3.5.3 外部名の宣言

他のモジュールとの間で相互に参照するシンボルを宣言します。この疑似命令には以下の3種類があります。

.globl (.global)	**グローバルシンボルの宣言**
.xdef (.public, .entry)	**外部定義名の宣言**
.xref (.extrn, .external)	**外部参照名の宣言**

.globl (.global) グローバルシンボルの宣言

書式： .globl <シンボル> [,<シンボル> …]
.global <シンボル> [,<シンボル> …]

機能： <シンボル> が外部名であることを宣言します。

解説： <シンボル> が外部名であることを宣言します。ソースファイル中に，<シンボル> が定義されていると，そのシンボルは外部定義宣言されているものとして扱われ，<シンボル> が定義されていない場合は，そのシンボルは外部参照宣言されているものとして扱われます。
Fig. 3-4 では，どちらのプログラムでもシンボル **strcpy** は外部名として宣言されていますが，Program A ではシンボルが定義されているので外部定義として扱われ， Program B では外部参照として扱われます。したがって，Program B から Program A の **strcpy** を参照することになります。

```
Program A
        .globl strcpy
        bsr strcpy
        ....
```

外部参照

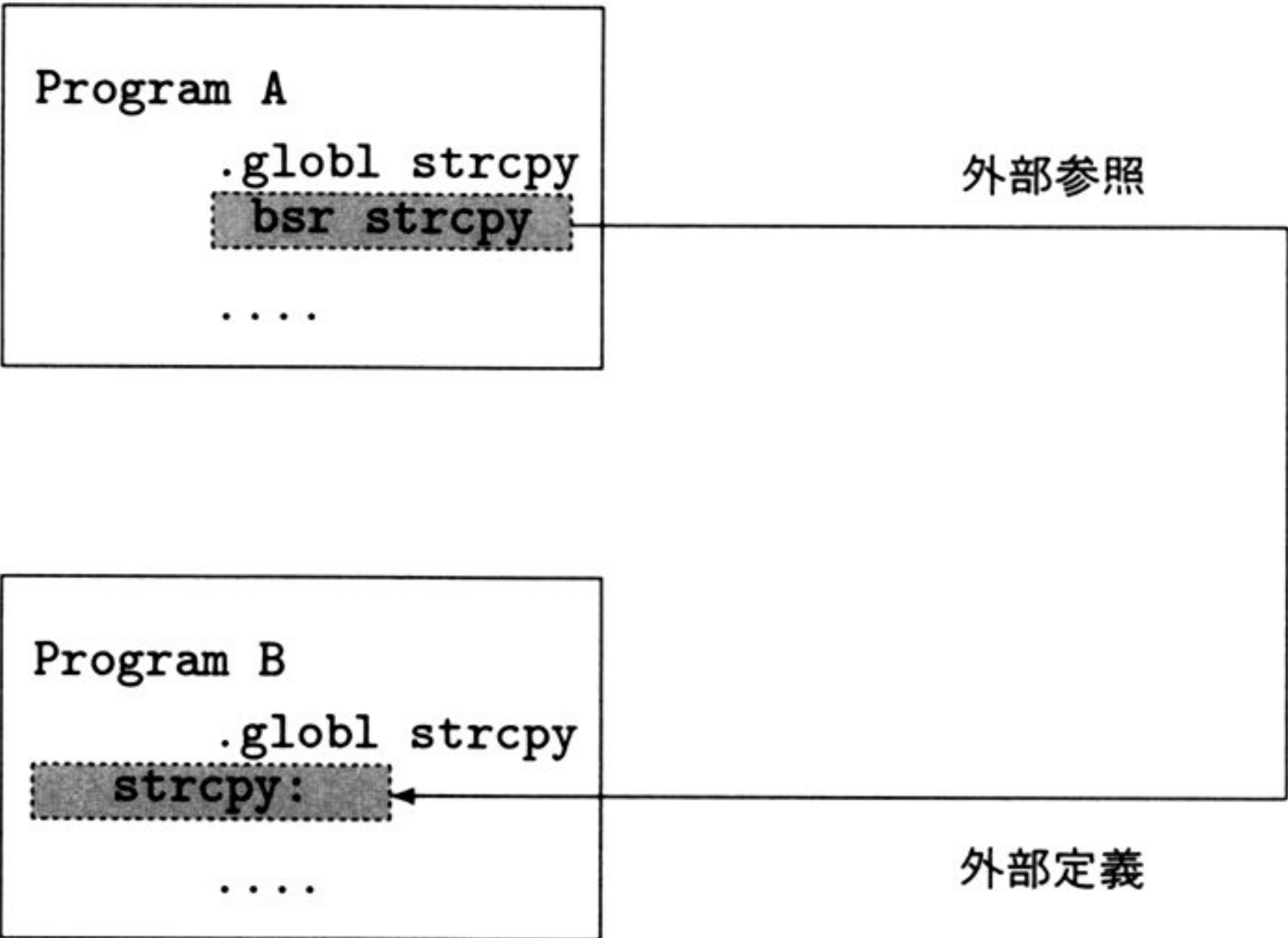
```
Program B
        .globl strcpy
strcpy:
        ....
```

外部定義

Fig. 3-4● グローバルシンボルと外部定義，外部参照

List 3-35 ● .globl 疑似命令

```
1:  * シンボル strcpy を外部名として宣言する
2:          .globl  strcpy
```

.xdef (.public/.entry) 外部定義名の宣言

書式： .xdef <シンボル> [,<シンボル> …]

.public <シンボル> [,<シンボル> …]

.entry <シンボル> [,<シンボル> …]

機能： <シンボル> が外部定義名であることを宣言します。

解説： <シンボル> が外部定義名であることを宣言します。これらの命令によって，<シンボル> を他のモジュールから参照することが可能になります。

List 3-36 ● .xdef 疑似命令

```
1:  * シンボル strlen を外部定義名として宣言する
2:          .xdef   strlen
```

.xref (.extrn/.external) 外部参照名の宣言

書式： .xref <シンボル> [,<シンボル> …]

.extrn <シンボル> [,<シンボル> …]

.external <シンボル> [,<シンボル> …]

機能： <シンボル> が外部参照名であることを宣言します。

解説： <シンボル> が外部参照名であることを宣言します。ソースファイル中の <シンボル> の参照は，すべて外部のモジュールに対して行われるようになります。

List 3-37 ● .xref 疑似命令

```
1:  * シンボル strcat を外部参照名として宣言する
2:          .xdef   strcat
```

3.5.4 シンボルの定義

シンボルに対して値を定義します。この疑似命令には以下の3種類があります。

.equ	**不変シンボル値の定義**
.set (=)	**可変シンボル値の定義**
.reg	**レジスタリストの定義**

.equ 不変シンボル値の定義

書式： <シンボル> .equ <式>

機能： 不変シンボルの値を定義します。

解説： オペランドフィールドの <式> の値を，ラベルフィールドの <シンボル> に割り当てます。この定義はソース全体に対して有効であり，他の値に再定義することはできません。
<式> には未定義のシンボルや，この式の後に定義されるシンボルは使えません。

List 3-38● .equ 疑似命令

```
1: * シンボル NAMELEN に定数 18 を定義する
2: NAMELEN equ      18
```

.set (=) 可変シンボル値の定義

書式： <シンボル> .set <式>
<シンボル> = <式>

機能： シンボルに一時的な値を割り当てます。

解説： オペランドフィールドの <式> の値を，ラベルフィールドの <シンボル> に一時的に割り当てます。.equ 疑似命令による定義と異なり，この定義はソースファイル中で何度でも変更できます。また，その際に前の定義を参照することもできます。
<式> には未定義のシンボルや，この式の後に定義されるシンボルは使えません。

List 3-39● .set 疑似命令

```
1:  VALUE    .set     10
2:           ...              シンボル VALUE の値は 10
3:           ...
4:  VALUE    .set     20
5:           ...              ここから先は，シンボル VALUE の値は 20 になる
6:           ...
```

.reg　レジスタリストの定義

書式：　<シンボル> .reg <レジスタリスト>

機能：　シンボルにレジスタリストを定義します。

解説：　MOVEM 命令のオペランドとして使用されるレジスタリストの内容を，<シンボル> に割り当てます。以後，レジスタリストは定義されたシンボルによって指定することができます。

List 3-40● .reg 疑似命令

```
1:  reglist .reg     d0-d7/a0-a6
2:  * reglist は d0-d7/a0-a6 と同じ
3:          movem.l reglist,-(sp)
```

3.5.5 マクロ制御

マクロの定義とそれに関する情報をアセンブラに与えます。この疑似命令には以下の7種類があります。

.macro	**マクロ定義の開始**
.local	**マクロ定義ブロック内の局所的シンボルの定義**
.endm	**マクロ定義の終了**
.exitm	**マクロ展開の打ち切り**
.rept	**繰り返しの開始** HAS拡張
.irp	**不定回数の繰り返しの開始** HAS拡張
.irpc	**不定回数の文字繰り返しの開始** HAS拡張

.macro マクロ定義の開始

書式： <シンボル> .macro [<仮引数> [,<仮引数> …]]

機能： マクロ定義を開始します。

解説： この疑似命令から.endm 疑似命令までの行の内容をマクロとして <シンボル> に割り当てます。

<仮引数> には，マクロ命令を展開する際の引数に対応する仮引数の名前を指定します。マクロ命令を展開するとき，マクロ定義された行の中の <仮引数> と同じ語に対応する引数が割り当てられます。

List 3-41 のように定義されたマクロ print に対して，List 3-42 のように引数 “msg(pc)” を与えます (1 行目)。すると仮引数である “string” が実際の引数と置き換えられて，3 ～ 5 行目のように展開されます。

List 3-41 ● マクロ定義

```
1:  print   .macro   string
2:          pea.l    string
3:          .dc.w    $ff09
4:          addq.l   #4,sp
5:          .endm
```

List 3-42 ● マクロの展開

```
1:          print    msg(pc)
2:                   ↓
3:          pea.l    msg(pc)
4:          .dc.w    $ff09
5:          addq.l   #4,sp
```

.local マクロ定義ブロック内の局所的シンボルの定義

書式： `.local <シンボル> [,<シンボル> … ]`

機能： マクロ定義ブロック内の局所的シンボルを定義します。

解説： マクロ定義ブロックの中だけで有効な，局所的なシンボルを定義します。ここで定義されたシンボルは，マクロ展開を行う際に他のシンボル定義と重ならないような局所的な名前[8]に変換されます。
List 3-43 のシンボル `loop` は局所的シンボルとして定義されているので，展開時には他のシンボルと重ならないような名前に変換されます。`.local` 疑似命令による定義がないと，このマクロを 2 回以上展開したときに，シンボルの二重定義でエラーが発生します。

8) "??xxxx" … xxxx は 4 桁の 16 進数です。

List 3-43 ● .local 疑似命令

```
1:  clear   .macro
2:          .local  loop
3:          moveq.l #10-1,d0
4:  loop:
5:          clr.l   (a0)+
6:          dbra    d0,loop
7:          .endm
```

.endm マクロ定義の終了

書式： `.endm`

機能： マクロ定義を終了します。

解説： マクロ定義を終了します。繰り返し疑似命令の定義を終了する際にも使われます。

List 3-44 ● .endm 疑似命令

```
1:  print   .macro  string
2:          pea.l   string
3:          .dc.w   $ff09
4:          addq.l  #4,sp
5:          .endm
```

.exitm マクロ展開の打ち切り

書式： `.exitm`

機能： マクロ定義の展開を途中で打ち切ります。

解説： マクロ命令を展開する際に，その展開を途中で打ち切ります。おもに条件つきアセンブル疑似命令と組み合わせて，ある条件が成立した際にその後のマクロ展開を行わないようにするために使用します。

List 3-45 ではマクロ展開の際，`.if` 疑似命令の条件が成立すると 4 行目の `.exitm` 疑似命令によって展開が打ち切られます。シンボル `final` の値が真 (0 以外) の場合に，6 行目の BRA 命令が展開されなくなります。

List 3-45 ● .exitm 疑似命令

```
1:  ent     .macro  adr
2:          lea.l   adr,a2
3:          .if     final
4:          .exitm
5:          .endif
6:          bra     label
7:          .endm
```

.rept　繰り返しの開始　HAS 拡張

書式： `.rept <式>`

機能： 命令の繰り返しを開始します。

解説： この疑似命令から `.endm` 疑似命令までの行の内容を，<式> の値の回数だけ繰り返します。<式> の値が 0 である場合は，繰り返しは一度も行われません。

<式> の値は定数でなければならず，前方参照値や外部参照値，アドレス値などは使えません。

以下の 2 つのプログラム (List 3-46 , List 3-47) は同じ意味になります。

List 3-46 ● .rept 疑似命令

```
1:          .rept   3
2:          move.l  (a0)+,(a1)+
3:          .endm
```

List 3-47 ● List 3-46 と等価な例

```
1:          move.l  (a0)+,(a1)+
2:          move.l  (a0)+,(a1)+
3:          move.l  (a0)+,(a1)+
```

.irp　不定回数の繰り返しの開始　HAS 拡張

書式：　.irp <仮引数>,<実引数> [,<実引数> …]

機能：　命令の不定回数の繰り返しを開始します。

解説：　この疑似命令から .endm 疑似命令までの行の内容を，<実引数> の個数だけ繰り返します。その際，繰り返す行の中にある <仮引数> と同じ語に，<実引数> の内容が 1 つずつ割り当てられます。
List 3-48 と List 3-49 は同じ意味になります。

List 3-48● .irp 疑似命令

```
1:        .irp    adr,com1,com2
2:        subq.w  #1,d0
3:        beq     adr
4:        .endm
```

List 3-49● List 3-48 と等価な例

```
1:        subq.w  #1,d0
2:        beq     com1
3:        subq.w  #1,d0
4:        beq     com2
```

.irpc　不定回数の文字繰り返しの開始　HAS 拡張

書式：　.irpc <仮引数>,<文字列>

機能：　命令の不定回数の文字繰り返しを開始します。

解説：　この疑似命令から.endm 疑似命令までの行の内容を，<文字列> の長さだけ繰り返します。その際，繰り返す行の中にある <仮引数> と同じ語に，<文字列> 中の文字が 1 文字ずつ割り当てられます。
List 3-50 と List 3-51 は同じ意味になります。

List 3-50● .irpc 疑似命令

```
1:        .irpc   ch,AB
2:        cmpi.b  #"&ch",d0
3:        beq     switch_&ch
4:        .endm
```

List 3-51● List 3-50 と等価な例

```
1:        cmpi.b  #"A",d0
2:        beq     switch_A
3:        cmpi.b  #"B",d0
4:        beq     switch_B
```

3.5.6 データの定義

プログラムのデータ領域の定義や確保を行います。この疑似命令には以下の 4 種類があります。

.dc	**定数データの定義**
.dcb	**定数ブロックの定義**
.ds	**メモリ領域の確保**
.even	**偶数境界の調整**

.dc 定数データの定義

書式： .dc[.<サイズ>] <式> [,<式> …]

機能： 定数データを定義します。

解説： メモリに定数データを定義します。 <式> はカンマ (,) で区切って複数指定することができます。

<サイズ> にはバイト (b)，ワード (w)，ロングワード (l) のいずれか 1 つを指定することができ，<式> が数値である場合は，その値が <サイズ> に従った大きさで格納されます。

また <式> に，アポストロフィ (') またはダブルクォーテーション (") で囲まれた文字列を指定した場合には，<サイズ> の指定によって次のように動作します。

- **バイト** (.dc.b)
  文字列は先頭から 1 文字ずつ ASCII コードに変換されて格納される
- **ワード** (.dc.w)
  バイトと同様に，1 文字ずつ ASCII コードに変換されるが，文字列の長さが奇数バイトの場合には，最後のワードの下位に文字が格納され，上位には 0 が入る
- **ロングワード** (.dc.l)
  バイトと同様に，1 文字ずつ ASCII コードに変換されるが，文字列の長さが 4 の整数倍に満たない場合には，最後のロングワードの下位に文字列が格納され，上位には 0 が入る

アセンブラは，自動的な偶数境界の調整などはいっさい行いません。したがって <サイズ> にバイトの指定をしたとき， <式> を奇数個記述したり， <式> に奇数バイトの文字列を指定したりすると，次のステートメントのアドレスが奇数になることがあるので注意してください。

拡張： **X680x0 HAS** では浮動小数点実数に対応したため，指定できる<サイズ>が拡張されています。

List 3-52 ● .dc.w 疑似命令

```
1:  * ワードサイズの定数データ 1, 2, 3 を定義する
2:          .dc.w 1,2,3
```

List 3-53 ● .dc.b 疑似命令

```
1:  * バイトサイズの定数データ $61, $62, $63 を定義する
2:          .dc.b   "abc"
```

List 3-54 ● .dc.w 疑似命令

```
1:  * ワードサイズの定数データ $6162, $0063 を定義する
2:          .dc.w   "abc"
```

List 3-55 ● .dc.l 疑似命令

```
1:  * ロングワードサイズの定数データ $00616263 を定義する
2:          .dc.l   "abc"
```

.dcb　定数ブロックの定義

書式： .dcb[.<サイズ>] <長さ>,<式>

機能： 定数ブロックを定義します。

解説： メモリに定数ブロックを定義します。<サイズ> にはバイト (b)，ワード (w)，ロングワード (l) のいずれか 1 つを指定することができ，メモリ領域は <長さ> で指定した個数の<サイズ> に従った <式> 値で満たされます。

<長さ> の値は定数でなければならず，前方参照値や外部参照値，アドレス値などは使えません。

拡張： **X680x0 HAS** では浮動小数点実数に対応したため，指定できる<サイズ>が拡張されています。

List 3-56 ● .dcb 疑似命令

```
1:  * ワードサイズ $22D8 を 16 個分定義する
2:          .dcb.w  16,$22d8
```

.ds　メモリ領域の確保

書式：　.ds[.<サイズ>] <長さ>

機能：　メモリ領域を確保します。

解説：　メモリ領域を確保します。<サイズ> にはバイト (b)，ワード (w)，ロングワード (l) のいずれか1つを指定することができ，指定した <サイズ> の値を <長さ> の個数だけ格納できるだけのメモリ領域を確保します。確保したメモリ領域の内容は初期化されません。

<長さ> の値は定数でなければならず，前方参照値や外部参照値，アドレス値などは使えません。

拡張：　**X680x0 HAS** では浮動小数点実数に対応したため，指定できる<サイズ>が拡張されています。

List 3-57● .ds 疑似命令

```
1:  * ロングワードサイズを 5 個分格納できる領域を確保する
2:          .ds.l   5
```

.even　偶数境界の調整

書式：　.even

機能：　偶数境界の調整を行います。

解説：　ロケーションカウンタの値が奇数の場合に，ロケーションカウンタを1だけ進めることで偶数境界を調整し，次のアドレスが必ず偶数になるようにします。

List 3-58● .even 疑似命令

```
1:  * .dc.b 疑似命令によって奇数になったロケーションカウンタ値を,
2:  * .even 疑似命令によって偶数に調整している
3:          .dc.b   0
4:          .even
5:          .dc.w   2
```

3.5.7 条件つきアセンブル

指定した条件によって，ソースプログラムのある範囲のアセンブルを行うかどうかを決定します。この疑似命令には以下の 7 種類があります。

`.if (.ifne)`	**条件が真のときにアセンブル実行**
`.iff (.ifeq)`	**条件が偽のときにアセンブル実行**
`.ifdef`	**シンボルが定義されているときにアセンブル実行**
`.ifndef`	**シンボルが定義されていないときにアセンブル実行**
`.else`	**反対の条件でアセンブル実行**
`.elseif`	**反対の条件が成立し，かつ指定の条件が真のときにアセンブル実行**
`.endif (.endc)`	**条件つきアセンブルの終了**

.if (.ifne) 条件つきアセンブル
.iff (.ifeq)
.ifdef / .ifndef
.else / .elseif

書式：

`.if <式>`
`.ifne <式>` ……… <式>が真のときにアセンブル実行

`.iff <式>`
`.ifeq <式>` ……… <式>が偽のときにアセンブル実行

`.ifdef <シンボル>` …… <シンボル>が定義されているときにアセンブル実行

`.ifndef <シンボル>` …… <シンボル>が定義されていないときにアセンブル実行

`.else` ………………… 反対の条件でアセンブル実行

`.elseif <式>` ………… 反対の条件が成立し，かつ <式>が真のときにアセンブル実行

機能：条件が成り立つときにアセンブルを実行します。

解説：指定した条件が成り立つかどうかによって，`.endif` 疑似命令との間にあるソースプログラムをアセンブルするかどうかを決定します。

<式> にはアセンブルを実行する条件を指定し，その値が 0 であれば偽，0 以外であれば真になります。<式> の値は定数でなければならず，前方参照値や外部参照値，アドレス値等は使えません。

List 3-59 では，シンボル `DEBUG` が定義されていたら `.else` 疑似命令までの行を，定義されていなかったら `.else` 疑似命令から `.endif` 疑似命令までの行をアセンブルします。

List 3-59● 条件つきアセンブルの実行

```
1:          .ifdef  DEBUG
2:  begin   equ     work
3:          .else
4:  begin   equ     $e00000
5:          .endif
```

.endif (.endc) 条件つきアセンブルの終了

書式： .endif

.endc

機能： 条件つきアセンブルを終了させます。

解説： 条件つきアセンブルを終了させます。

List 3-60● 条件つきアセンブルの終了

```
1:          .ifdef  DEBUG
2:  begin   equ     work
3:          .else
4:  begin   equ     $e00000
5:          .endif
```

3.5.8 リストファイル制御

アセンブルリストの出力に関する制御を行います。この疑似命令には以下の 8 種類があります。

.list	**アセンブルリストの出力**
.nlist	**アセンブルリストの出力抑制**
.width	**アセンブルリストの表示桁数の指定**
.page	**アセンブルリストの改ページ，ページ長指定**
.title	**アセンブルリストのタイトル指定**
.subttl	**アセンブルリストのサブタイトル指定**
.lall	**マクロ展開行の出力**
.sall	**マクロ展開行の出力抑制**

.list　アセンブルリストの出力

書式： .list

機能： リストファイル作成時に，リストの出力を指示します。

解説： リストファイル作成時に，アセンブラに対してリスト出力の実行を指示します。アセンブル開始時は，デフォルトとしてこの状態になっています。

.nlist　アセンブルリストの出力抑制

書式： .nlist

機能： リストファイル作成時に，リスト出力を行わないようにします。

解説： リストファイル作成時，.list 疑似命令のある行までのリスト出力を抑制します。

List 3-61 では.nlist 疑似命令から .list 疑似命令までの行は，リストファイルに出力されません。

List 3-61 ● .list 疑似命令

```
1:          .nlist
2:          .include header.h
3:          .list
```

.width　アセンブルリストの表示桁数の指定

書式：　.width <式>

機能：　リストファイル作成時の表示桁数を指定します。

解説：　リストファイル作成時の表示桁数を <式> の値に設定します。<式> には 80 ～ 248 の 8 の倍数を指定することができ，デフォルトでは 136 桁が指定されています。リストファイルで指定の桁数を超える行は，複数の行に分割されて出力されます。

List 3-62 ● .width 疑似命令

```
1:  * リストファイルの表示桁数を 80 桁に設定する
2:          .width  80
```

.page　アセンブルリストの改ページ，ページ長指定

書式：
1. .page　　改ページ指示 (マイナーページ番号を増加)
2. .page +　　改ページ指示 (メジャーページ番号を増加) **HAS 拡張**
3. .page <式>　ページ長指定 **HAS 拡張**

機能：　リストファイルの改ページ指示やページ長の指定を行います。

解説：
1. リストファイルをこの位置で改ページします。リストファイルの各ページには，ページ番号が “<メジャー> – <マイナー>” の形で出力されていますが，この命令はマイナーページ番号を 1 つ増加します。
2. 1. と同様に改ページを行いますが，メジャーページ番号を 1 つ増加し，マイナーページ番号を 1 にリセットします。
3. リストファイルの 1 ページの行数 (ページ長) を <式> の値に設定します。<式> には 10 ～ 255 の値を指定することができ，デフォルトでは 56 行が指定されています。リストファイルは，指定の行数だけ出力すると，改ページの指示がなくとも改ページを行います。

List 3-63 では，リストファイルのページ長を 66 行に設定しています。

List 3-63 ● .page 疑似命令

```
1:          .page   66
```

.title　アセンブルリストのタイトル指定

書式：　.title <文字列>

機能：　アセンブルリストのタイトルを指定します。

解説：　リストファイルの各ページの最初に出力されるタイトルを，<文字列>に設定します。<文字列> は 1 行以内で記述します。一度設定したタイトル文字列は，リストファイルの全体にわたって有効になります。

List 3-64 ● .title 疑似命令

```
1:  * タイトル文字列を，"sample program main routine" に設定する
2:          .title  sample program main routine
```

.subttl　アセンブルリストのサブタイトル指定

書式：　.subttl <文字列>

機能：　アセンブルリストのサブタイトルを指定します。

解説：　リストファイルの各ページの最初に出力されるサブタイトルを，<文字列> に設定します。<文字列> は 1 行以内で記述します。サブタイトルは，リストファイル中で何度も変更することができます。
設定されたタイトル，サブタイトルは，各ページの先頭に次のように出力されます。

```
X68k High-speed Assembler v2.5x Copyright 1990,91,92 by Y.Nakamura
sample program main routine                          mnt/dd/yy hh:mm:ss
initialize part                                      Page M-m
```

List 3-65 ● .subttl 疑似命令

```
1:  * サブタイトルを "initialize part" に設定する
2:          .subttl initialize part
```

.lall　マクロ展開行の出力

書式：　.lall

機能：　リストファイル作成時に，マクロ展開行の出力を指示します。

解説：　リストファイル作成時，.lall 疑似命令以降の行でマクロ展開が行われたら，マクロ行の次に実際の展開内容を出力するようにします。
たとえば，List 3-66 のソースプログラムをアセンブルしてリストファイルを作成すると，List 3-67 のようにマクロの展開内容が出力されます。

List 3-66 ● .lall 疑似命令

```
 1:  print   .macro   string
 2:          pea.l    string
 3:          .dc.w    $ff09
 4:          addq.l   #4,sp
 5:          .endm
 6:          ....
 7:          ....
 8:          .lall
 9:          print    msg(pc)
10:          rts
```

List 3-67 ● List 3-66 のアセンブルリスト

```
1:   8 00000000                .lall
2:   9 00000000                print    msg(pc)
3:   9 00000000*487A????       pea.l    msg(pc)
4:   9 00000004*FF09           .dc.w    $ff09
5:   9 00000006*588F           addq.l   #4,sp
6:  10 00000008 4E75           rts
```

.sall　マクロ展開行の出力抑制

書式： .sall

機能： リストファイル作成時に，マクロ展開行を出力しないように指示します。

解説： リストファイル作成時に，マクロ展開が行われても，その内容を出力しないようにします。アセンブル開始時は，この状態がデフォルトになっています。

たとえば，List 3-68 のソースプログラムをアセンブルしてリストファイルを作成すると，List 3-69 のようにマクロの展開内容は出力されなくなります。

List 3-68● .sall 疑似命令

```
 1:  print    .macro  string
 2:           pea.l   string
 3:           .dc.w   $ff09
 4:           addq.l  #4,sp
 5:           .endm
 6:           ....
 7:           ....
 8:           .sall
 9:           print   msg(pc)
10:           rts
```

List 3-69● List 3-68 のアセンブルリスト

```
1:   8 00000000                    .sall
2:   9 00000000                    print   msg(pc)
3:  10 00000008 4E75               rts
```

3.5.9 シンボリックデバッグ情報の指定

9)SHARPのXCに付属する純正ソースコードデバッガです。

GDBやSCD[9)]などによってソースコードデバッグを行うために，オブジェクトファイルに出力するソースファイルのシンボリックデバッグ情報を指定します。この疑似命令には以下の3種類があります。

.file	**ソースファイル名の出力指定**
.ln	**行番号とロケーションの対応の出力指定**
.def ～ .endef	**シンボルテーブルエントリの作成**

以下の疑似命令は，.def ～ .endef 疑似命令によって作成するシンボルテーブルエントリにシンボルの情報を指定するための疑似命令で，7種類あります。

.val	**シンボルの値の指定**
.scl	**記憶クラスの宣言**
.type	**C言語における型の宣言**
.tag	**タグ名の宣言**
.line	**行番号の指定**
.size	**サイズの指定**
.dim	**配列の指定**

.file　ソースファイル名の出力指定

書式： .file "<ファイル名>"

機能： ソースファイル名をオブジェクトファイルに出力します。

解説： ソースファイル名を，オブジェクトファイルに出力します。<ファイル名>には，C言語のソースファイル名を指定します。
List 3-70では，ファイル名“sample.c”をオブジェクトファイルに出力しています。

List 3-70 ● .file疑似命令

```
1:          .file   "sample.c"
```

.ln　行番号とロケーションの対応の出力指定

書式： .ln <行番号> [,<ロケーション値>]

機能： 行番号とロケーションの対応を，オブジェクトファイルに出力します。

解説： 行番号とロケーションの対応を，オブジェクトファイルに出力します。
<行番号> には，C 言語で記述されたソースプログラムの各関数の開始行からの相対的な行番号を指定します。<ロケーション値> には，行番号に対応する機械語命令の存在するロケーション値を指定します。省略すると，この疑似命令が存在する位置のロケーション値が指定されたものとします。
List 3-71 では，現在のロケーション値に行番号 1 を対応させています。行番号は，List 3-72 のように関数の開始行からの相対値で表記します。

List 3-71 ● .ln 疑似命令

```
1:           .ln      1
```

List 3-72 ● .ln 疑似命令とソースの対応

```
1:  void main(int argc, char *argv[])
2:  {                                          ←.ln 1
3:      printf("%s\n", argv[0]);               ←.ln 2
4:  }                                          ←.ln 3
5:
6:  int negate(int a)
7:  {                                          ←.ln 1
8:      return -a;                             ←.ln 2
9:  }                                          ←.ln 3
```

.def ～ .endef　シンボルテーブルエントリの作成

書式： .def <シンボル名>
属性を指定する疑似命令
…
…
.endef

機能： シンボルテーブルエントリを作成します。

解説： C 言語のソースプログラム中のシンボル名に対して，その属性を与えるためのシンボルテーブルエントリを作成します。.def 疑似命令と .endef 疑似命令との間にある，シンボルの属性を指定する疑似命令をもとにシンボルテーブルが作成されます。
<シンボル名> には，シンボルテーブルエントリを作成するシンボルの名前を指定します。この名前は，C 言語のプログラムがコンパイルされたアセンブラソース上での名前となります。また，C 言語のプログラム構造をオブジェクトファイルに出力するために，ピリオド (.) で開始する以下のような特殊なシンボル名を使用します。

- .bf　　関数の始まりを宣言するシンボル

- .ef　関数の終わりを宣言するシンボル
- .bb　関数内ブロックの始まりを宣言するシンボル
- .eb　関数内ブロックの終わりを宣言するシンボル
- .eos　構造体/共用体/列挙宣言の終了を宣言するシンボル

List 3-73は，配列 static char (*foo[10])() のシンボル _foo のシンボルテーブルエントリを作成します。

List 3-73● .def ~ .endef疑似命令

```
1:        .def    _foo
2:        .val    _foo
3:        .scl    3
4:        .dim    10
5:        .size   40
6:        .type   626
7:        .endef
```

.val　シンボルの値の指定

書式：　.val <式>

　　　　.val .

機能：　式の値をシンボルのものとします。

解説：　シンボルテーブルエントリで属性を指定する疑似命令です。シンボルに対して式の値を与えます。

<式>には，シンボルが関数ならばそのロケーションを，シンボルがauto変数ならばスタック上のオフセット値を，構造体のメンバ名ならば構造体先頭からのオフセット値を指定します。また，<式>の代わりに“.”を指定することで，現在のロケーション値を与えることができます。

List 3-74のシンボルの値は，アセンブラソースのシンボル _foo の値となります。

List 3-74● .val疑似命令

```
1:        .val    _foo
```

.scl　記憶クラスの宣言

書式：　.scl <式>

機能：　シンボルの記憶クラスを宣言します。

解説： シンボルテーブルエントリで属性を指定する疑似命令です。シンボルに対して記憶クラスを与えます。<式> の値と記憶クラスは，Table 3-2 のように対応しています。

Table 3-2● 記憶クラスと数値との対応

数値	記憶クラス
1	auto 変数
2	extern 変数/関数
3	static 変数/関数
4	register 変数
8	構造体のメンバ名
9	関数の引数
10	構造体のタグ名
11	共用体のメンバ名
12	共用体のタグ名
13	typedef された型の名前
15	列挙タグ名
16	列挙メンバ名
18	構造体/共用体のビットフィールド
100	関数内ブロックの開始/終了 (.bb/.eb で使用)
101	関数の開始/終了 (.bf/.ef で使用)
102	構造体/共用体宣言の終了 (.eos で使用)
-1	関数定義の終了

List 3-75● .scl 疑似命令

```
1:  * シンボルのクラスは static 変数/関数となる
2:                  .scl    3
```

.type C 言語における型の宣言

書式： .type <式>

機能： シンボルの C 言語における型を宣言します。

解説： シンボルテーブルエントリで属性を指定する疑似命令です。シンボルに対してその型を与えます。シンボルの型は，<式> の値によって次のように決定されます。

<式> の値は 16 ビットの無符号数値として扱います。このうち，最下位の 4 ビットによって，シンボルの基本型が決定されます。最下位 4 ビットと基本型の対応は Table 3-3 のとおりです[10)]。

シンボルの型が派生型である場合は，基本型の 4 ビットの上位に 2 ビットずつ，Table 3-4 のような派生型の情報が付加します。このとき，基本型に対しての優先順位が高いほど，付加するビットは下位になります。

10)ここでは便宜上，2 進数で表記します。

List 3-76では，626 = $0272 = %10_01_11_0010 ですので，基本型は char になります。また派生型は，優先順位の高い順に配列，ポインタ，関数になるので，次のようになります。

基本型 ………………………… char foo
＋派生型 （～の配列） ………… char foo[]
＋派生型 （～へのポインタ） …… char *foo[]
＋派生型 （～を返す関数） ……… char (*foo[])()

したがって，List 3-76の型情報の示す型は，シンボルを foo と仮定すると，char(*foo[])()[11]になります。

11) char を返す関数へのポインタの配列です。

Table 3-3● 基本型と数値の対応

数　値	基本型
%0000	void
%0010	char
%0011	short int
%0100	long int
%0110	float
%0111	double
%1000	struct
%1001	union
%1010	列挙タグ名
%1011	列挙メンバ名
%1100	unsigned char
%1101	unsigned short int
%1110	unsigned long int

Table 3-4● 派生型と数値の対応

数値	派生型
%01	～へのポインタ
%10	～を返す関数
%11	～の配列

List 3-76● .type 疑似命令

```
1:          .type   626
```

.tag　タグ名の宣言

書式： .tag <タグ名>

機能： シンボルで使用するタグ名を宣言します。

解説： シンボルテーブルエントリで属性を指定する疑似命令です。シンボルが構造体や共用体，列挙型であった場合に，そのシンボルの構造を定義しているタグのタグ名を宣言します。

<タグ名> は，タグ名という記憶クラスをもったシンボルテーブルエントリとして宣言されている必要があります。

List 3-77 では，シンボルの使用するタグは _.fake0 となります。

List 3-77● .tag 疑似命令

```
1:          .tag    _.fake0
```

.line　行番号の指定

書式： .line <式>

機能： シンボルに行番号や行数を与えます。

解説： シンボルテーブルエントリで属性を指定する疑似命令です。C 言語のプログラム構造を表現するための特殊なシンボルに対して，<式> によってその構造の開始する行番号や行数を与えます。各シンボルによって，<式> の値の意味は次のように異なります。

- .bf　C 言語のソースプログラムで，関数が開始している行番号を指定する
- .ef　関数本体のソースプログラムでの行数を指定する
- .bb　関数の開始する行から見た，関数内ブロックの開始する相対的な行番号を指定する
- .eb　関数の開始する行から見た，関数内ブロックの終了する相対的な行番号を指定する

List 3-78● .line 疑似命令

```
1:  * 関数本体は行番号 3 から開始する
2:          .def    .bf
3:          .val    .
4:          .scl    101
5:          .line   3
6:          .endef
```

.size　サイズの指定

書式： .size <式>

機能： シンボルのサイズを宣言します。

解説： シンボルテーブルエントリで属性を指定する疑似命令です。シンボルに対して，そのサイズを与えます。

<式> には，シンボルが配列や構造体／共用体である場合は，そのシンボルがメモリ上に占めるバイト数，ビットフィールドである場合は使用するビット数を指定します。

List 3-79● .size疑似命令

```
1:  * シンボルのサイズは 40 バイト
2:            .size   40
```

.dim　配列の指定

書式： .dim <式1> [,<式2>, … ,<式4>]

機能： 配列のシンボルの要素数と次元を宣言します。

解説： シンボルテーブルエントリで属性を指定する疑似命令です。配列として宣言されたシンボルに対して，その要素数と次元を与えます。

<式> には配列の要素数を指定します。<式> は “,” で区切って，最大4つまで指定することができ，それによって最大4次元までの配列を表すことができます。

List 3-80● .dim疑似命令

```
1:  * シンボルは 1 次元で要素数 10 の配列
2:            .dim    10
```

3.6 ………………………………… HASの制限

最後に，HASの仕様上の制限をあげておきます。

❖ **行の長さ**

ソースファイルやインクルードファイルの 1 行は 255 文字までです[1]。

❖ **シンボルの識別長**

シンボルは，その長さのすべてを識別します。実際には行の長さの制限があるので，識別長もその制限に依存します。

❖ **シンボル数**

アセンブル時に扱えるシンボル数は，最大 32767 までです[2]。デフォルトでは 10000 シンボルまでに制限されていますが，コマンドライン上で "-m" スイッチ[3]によって，その上限を変更することができます。
シンボルは定義のたびにメモリ中にその内容を格納しますので，実際にはシンボル数はメモリ容量にも依存します。

❖ **インクルードファイル**

インクルードファイルは 8 重までネスティング可能です。

❖ **マクロ**

マクロ展開は 32 重までネスティング可能です。マクロは定義のたびにメモリ中にその定義内容を格納しますので，マクロ定義の長さはメモリ容量に依存します。

1)X680x0 HAS では，1 行の長さは 1023 文字までに拡張されています。

2)X680x0 HAS では，この制限はなくなっています。アセンブル時にはメモリの許す限りのシンボルを扱えます。

3)最大シンボル数の指定です。

Chapter 4

X68000 HLK

High–Speed Linker (以下 HLK と略記) は，SHARP の X68000 用純正リンカ LK.X (以下 LK と略記) とほぼコンパチブルなリンカです。また，HLK は LK の機能に加えて，いくつかの拡張機能が追加されています。

本章では，コンパイラ，アセンブラで作成されたオブジェクトファイルをまとめて 1 つの実行ファイルを作成するコマンドであるリンカ (HLK) の使用方法やトラブルシューティングなどを紹介します。

4.1 ……………………………………… HLKの概要

リンカは，アセンブラで作成されたオブジェクトファイルと，ライブラリファイルをまとめて1つの実行ファイルを作成します。オブジェクトファイルには，外部参照シンボルや外部定義シンボルの情報が入っています。外部参照シンボルは，それ自身ではシンボルの値を決定できませんので，同じ名前の外部定義シンボルの値が必要になります[1]。

1)「外部参照」を参照(第3.3.2節 P.88)。

リンカは，これらすべての外部参照シンボルが同じ名前をもつ外部定義シンボルと対応がとれるように，オブジェクトファイルをリンクして実行ファイルを作成します。

4.1.1 HLKで使用するファイル

HLKではいくつかのファイルを使用します。ここでは，それらのファイルを入力ファイルと出力ファイルに分けて説明をします。

◆ 入力ファイル

次の3つのファイルは，HLKで実行ファイルを作成するときに，HLKに与えるファイルです。

❖ **オブジェクトファイル**

アセンブラソースをアセンブラに与えることによって作成されるファイルです。このファイルは，大まかにいって「外部参照シンボル」,「外部定義シンボル」,「シンボリックデバッグ情報」,「プログラム本体の情報」から構成されています。ファイルの拡張子は，“.o”です。

このファイルをリンカに与えることによって，実行ファイルが作成されます。

❖ **ライブラリファイル**

ライブラリファイルは，複数のオブジェクトファイルをまとめたファイルです。よく使用されるオブジェクトファイルをまとめておき，このファイルをリンカに渡せば，リンカは必要なオブジェクトファイルだけをリンクしてくれます。

ライブラリファイルには，アーカイブ形式[2]とライブラリアン形式[3]の2種類あります。アーカイブ形式はオブジェクトファイルをまとめて1つのファ

2)拡張子は，‘.a’です。

3)拡張子は，‘.l’です。

イルにしたもので，ライブラリアン形式はアーカイブ形式にシンボルのインデックスが付属している形になっています。

ライブラリファイルとオブジェクトファイルの相違点は，ライブラリファイルが与えられたとき，リンカはライブラリファイル中の必要なオブジェクトファイルをリンクするということです。オブジェクトファイルは，指定すれば必ずリンクされます。

極端な話，外部定義シンボルをもたないオブジェクトファイルで構成されているライブラリファイルは，リンカに与えても決してリンクされません。

❖ インダイレクトファイル

リンカは，コマンドラインからオブジェクトファイル名やオプション等を読み取る代わりに，インダイレクトファイルからも読み取ることができます。

インダイレクトファイルを使用することにより，255 文字を超えるようなコマンドラインが指定できたり，長いコマンドラインを指定する手間を省くことができます。

このファイルは，普通のテキストファイル形式です。スペース，タブ，改行コードはオプションやファイル名の区切り記号になります。

◆ 出力ファイル

次の 2 つのファイルは **HLK** が出力するファイルです。

❖ 実行ファイル

実行ファイルは，リンカに与えられたオブジェクトファイル，アーカイブファイルから作成されます。拡張子は，“.x” になります。実行ファイルには，プログラム本体や，プログラムをデバッグするための情報[4]が入っています。

4)シンボル情報やシンボリックデバッグ情報のことです。

❖ マップファイル

マップファイルは，オプションを指定することにより実行ファイルとともに作成されるファイルです。このファイルには **HLK** により作成された実行ファイルが，どのオブジェクトファイルから構成されているかや，ラベルの定義，参照情報など実行ファイルに関する情報が書かれています。List 4-1 にマップファイルの例を示して，このファイルの読み方を説明します。

List 4-1 ● マップファイルの例

```
 1:  ============================================================
 2:  prog.x
 3:  ============================================================
 4:  exec                          : 0000051c
 5:  text                          : 00000000 - 0000087f (00000880)
 6:  data                          : 00000880 - 00000995 (00000116)
 7:  bss                           : 00000996 - 000019fd (00001068)
 8:  common                        : 000019fe - 00001acb (000000ce)
 9:  stack                         :
10:  rdata                         : 00000000 - 0000001d (0000001e)
11:  rbss                          :
12:  rcommon                       :
13:  rstack                        :
14:  rldata                        :
15:  rlbss                         :
```

```
16:  rlcommon                   :
17:  rlstack                    :
18:
19:
20:  ==========================================================
21:  bar.o
22:  ==========================================================
23:  align                      : 00000002
24:  text                       : 00000000 - 000006ab (000006ac)
25:  data                       :
26:  bss                        : 00000996 - 00000a5d (000000c8)
27:  stack                      :
28:  rdata                      : 00000000 - 0000001d (0000001e)
29:  -------------------------- xref --------------------------
30:  symbol_from_foo            : in foo.o
31:  symbol_of_common           : in bar.o
32:  -------------------------- xdef --------------------------
33:  symbol_from_bar            : 00000000 (text    )
34:  -------------------------- comm --------------------------
35:  symbol_of_common           : 000019fe - 00001acb (000000ce)
36:
37:                             .
38:                             .
39:                             .
```

マップファイルは大きく分けて，実行ファイルの情報とオブジェクトファイルの情報の 2 つから構成されています。各項目について説明します。カッコ内の行番号は，List 4-1 での位置です。

❖ 実行ファイルの情報

- **実行ファイル名**　(1 ～ 3 行)
  実行ファイルの名前
- **実行アドレス**　(4 行)
  実行ファイルを起動したときに，最初に実行されるアドレス
- **各セクションの位置と大きさ**　(5 ～ 17 行)
  セクション名[5]と，そのセクションが占める範囲。カッコ内はセクションのサイズを表す。セクション名だけの行は，そのセクションが使用されていないことを示す

5)「セクション」を参照 (第 3.3.1 節 P.87)。

❖ オブジェクトファイルの情報

- **オブジェクトファイル名**　(20 ～ 22 行)
  オブジェクトファイルの名前。ライブラリ中のオブジェクトファイルの場合は，横にライブラリファイルの名前がつく
- **アラインメント値**[6]　(23 行)
  オブジェクトファイルの各セクションのアラインメント値
- **各セクションの位置と大きさ**　(24 ～ 28 行)
  セクション名と，そのセクションが占める範囲。カッコ内はセクションのサイズを表す。セクション名だけの行は，そのセクションが使用されていないことを示している。相対セクションは，使用されているセクションのみ情報が出力される

6) X680x0 HLK で追加されました。

- **外部参照シンボル** (29 〜 31 行)
  オブジェクトファイル中で外部参照しているシンボルのリスト。シンボル名と，外部参照しているシンボルが定義されているオブジェクトファイル名が出力される。**common** セクションのシンボルも出力されるが，これは仕様である
- **外部定義シンボル** (32 〜 33 行)
  オブジェクトファイル中で外部定義しているシンボルのリスト。シンボル名と外部定義しているシンボルの値と属性[7]が出力される
- **コモンエリア** (34 〜 35 行)
  オブジェクトファイル中で使用しているコモンエリア[8] のシンボルのリスト。シンボル名と，そのシンボルが占める範囲と大きさが出力される。大きさは，必ずしもそのファイルで定義した大きさにはならない。なぜならば，もっと大きい領域を定義しているオブジェクトファイルがあるかもしれないからである

7)「属性」を参照 (第 4.5.1 節 P.155)。

8)「共通データエリア (コモンエリア)」を参照 (第 3.3.3 節 P.89)。

4.1.2 HLK で使用する環境変数

基本的に **HLK** は環境変数を設定しなくても使用することができますが，次に示すような環境変数を設定することによって，より使用しやすくなります。なお，環境変数は大文字／小文字を区別するので注意してください。

❖ SILK (silk)

この環境変数 SILK またはsilk の内容は，**HLK** に指定されたコマンドラインの最後につけ加えられます。好みに応じて，自分がよく使うようなオプション[9] や，コンパイラドライバ[10]ではリンカに渡すことができないオプションを指定することができます。

もし環境変数 SILK が設定されている場合は，環境変数 silk の内容は参照されません。これら 2 つの環境変数が設定されていなくても，**HLK** は正常に動作します。

9)「起動方法と書式」を参照 (第 4.1.3 節 P.138)。

10)たとえば，gcc.x や cc.x などです。

❖ lib

この環境変数は，**HLK** に -l オプションを指定することで参照されるようになります。環境変数 lib にライブラリのディレクトリ名を設定しておくことで，ライブラリをリンクするときにフルパスで指定しなくても，ライブラリのファイル名だけですむようになるので，コマンドラインを短く，すっきりさせることができます。

環境変数 lib が設定されていない場合[11]は，-l オプションを使用すると，次の図のようなエラーメッセージが出力されます。また，-l オプションが有効でないため clib.a を見つけることもできません。そのため，さらに余分なエラーが発生します。

11)設定していない人は，あまりいないと思いますが。

```
A>set
A>
A>hlk main.o -l clib.a
Undefined environment variable 'lib'
Not found : clib.a
Undefined symbol(s) in a.o
__main

A>
```

4.1.3 起動方法と書式

HLK はコマンドラインから次のような書式で起動します。

HLK [<オプションスイッチ>] <ファイル名> [<ファイル名> ...]

<オプションスイッチ> を指定することにより **HLK** の動作を変更することができます。オプション文字は大文字／小文字の区別をしていません。LK ではオプションの先頭の文字は，“-”(ハイフン) もしくは “/”(スラッシュ) になっていましたが，**HLK** ではスラッシュをファイル名の一部とみなすようになっているので，オプションの先頭の文字は，“-” だけになります。

引数が必要なオプションの場合[12]，オプションと引数の間に空白文字を入れなくてもかまいません。オプションは，“**-ax**” のように複数のオプションをまとめて指定することはできません。この場合は，“-a -x” のように指定します。

ファイル名には，オブジェクトファイル，ライブラリファイルが指定できます。拡張子を省略した場合，“.o” を拡張子とみなします。またオブジェクトファイル名の先頭が “+” の場合は，“+” を取り除いたオブジェクトファイルを最初にリンクするようにします。オブジェクトファイルが複数指定された場合は，最後に指定されたオブジェクトファイルが先頭になります。

コマンドラインに何も指定しなかったり，オブジェクトファイルを指定しなかった場合は，簡単な説明とタイトルを表示します[13]。**HLK** のバージョンを確認したい場合は，コマンドラインに何も指定しないで起動すれば確認できます。

環境変数 SILK または silk が定義されていると，その内容がコマンドラインの最後につけ加えられます。よく使うオプションやコンパイルドライバでは，**HLK** に渡すことができないオプション[14]をつねに指定することができます。

以下のように設定すると **HLK** の起動時にタイトルを表示するようになります。

```
A>set SILK=-t
A>lk main.o
X68k SILK Hi-Speed Linker v2.** Copyright 1989-92 SALT
A>
```

12)たとえば，‘-i’ オプションなど。

13)HLK の名前が LK になっているのは，筆者が HLK.X を LK.X にリネームして使用しているためです。

14)たとえば，‘-t’ オプションなど。

4.1.4 オプションスイッチ

HLK で用意されているオプションを以下に示します[15]。

- “-a”　実行ファイルの拡張子省略時に “.x” をつけない
- “-d”　外部定義シンボルの登録
- “-i”　インダイレクトファイルの指定
- “-l”　ライブラリパスの使用
- “-m”　最大シンボル数の設定
- “-o”　実行ファイル名の指定[16]
- “-p”　マップファイルの作成
- “-s”　セクション情報を実行ファイルに埋め込む
- “-t”　起動時にタイトルを表示する
- “-v”　バーボーズモードの指定
- “-w”　ワーニングメッセージの抑制
- “-x”　シンボルテーブルの出力禁止
- “-z”　-v オプションを無効にする

15)各オプションの詳細は Vol.2 を参照してください。

16)従来はバグがありました。X680x0 HLK では，バグフィックスしています。

4.2 使用例

このセクションではリンカの使用例を示して，どのように実行ファイルを作成するのかを紹介します。カレントディレクトリは a:\ ，ライブラリがあるディレクトリは a:\lib\ とします。環境変数 lib には，a:\lib\ が設定されていて，環境変数 SILK，silk は設定されていないものとします。

- `hlk main`
  オブジェクトファイル main.o をリンクして，実行ファイル main.x を作成します。このとき，main.o に外部参照シンボルがあるとエラーになってしまいます。
- `hlk main.o sub1.o sub2.o`
  オブジェクトファイル main.o，sub1.o，sub2.o をリンクして，実行ファイル main.x を作成します。
- `hlk -o mytool main.o sub1.o`
  オブジェクトファイル main.o，sub1.o をリンクして，実行ファイル mytool.x を作成します。
- `hlk main.o a:\lib\libc.a`
  オブジェクトファイル main.o とライブラリファイル libc.a をリンクして，実行ファイル main.x を作成します。
- `hlk -o tool.x -i indirect`
  インダイレクトファイル[1]indirect に書かれているファイルをリンクして，実行ファイル tool.x を作成します。インダイレクトファイル indirect には，オブジェクトファイルとライブラリファイルの名前を書いておきます。
- `hlk main.o sub.o -l libc.a -p`
  オブジェクトファイル main.o，sub.o とライブラリファイル libc.a をカレントディレクトリから捜します。もし見つからなければ，a:\lib\ からファイルを捜します。見つけたファイルをリンクして，実行ファイル main.x とマップファイル[2]main.map を作成します。
- `hlk -s -o sxutil.x sxobjr.o a:\lib\libsx.a`
  オブジェクトファイル sxobjr.o とライブラリファイル libsx.a をリンクして，実行ファイル sxutil.x を作成します。-s オプションがついているのでセクション情報が埋め込まれています。GCC で OBJR 形式の実行ファイルを作成するときは，リンカにこのオプションを与えてください。

1)「インダイレクトファイル」を参照 (第 4.1.1.1 節 P.135)。

2)「マップファイル」を参照 (第 4.1.1.2 節 P.135)。

- **hlk third.o +second.o +first.o**

  オブジェクトファイルを **first.o**, **second.o**, **third.o** の順にリンクして，
  実行ファイル **first.x** を作成します。

4.3 ……………………………… HLK と LK の違い

HLK は LK の機能をほとんどすべて含んでいますが，リンク方法や仕様が多少異なります。ここでは，それらの違いについて解説します。普段は，それらの違いについてあまり気をつける必要はありません。

❖ オプション

LK にあったオプションで **HLK** にないオプションや，意味が変わってしまったオプションがあります。それらのオプションは Table 4-1 のとおりです。

Table 4-1 ● LK と意味の異なるオプション

オプション名	LK での動作	HLK での動作
-b	ベースアドレス指定	廃止（使用するとエラー）
-m	最大シンボル数	無効
-t	テンポラリパスの指定	起動時のタイトル表示

❖ リンク時間

純正の LK に比べて，1/2 ～ 1/7 の時間でリンクできます。ただし，すべてオンメモリ処理するために **HLK** に与えるファイルの 3 ～ 5 倍のメモリが必要です。

❖ シンボルの制限

LK では処理できるシンボル数の最大値を与えなければならなかったのですが，**HLK** ではメモリの許すかぎり処理できるようになっています。シンボル長についても，メモリの許すかぎり処理できるようになっています[1]。

LK との互換性のために -m オプションが使用できますが，使用しても **HLK** の動作に影響を与えないようになっています。

❖ エラーレポート

未定義な外部参照シンボル[2] が存在した場合に，どのファイルが外部参照していたかの情報や，エラーが発生した場合に，オブジェクトファイルのどの場所で発生したかの情報をレポートします。これにより，エラーの原因を発見しやすくなっています[3]。

❖ オプションの指定でシンボルを外部定義可能

HLK に与えるオプションから，シンボルを外部定義することができます。シンボルの属性[4] は絶対値になります。

1)LK でもシンボル長の制限はないと思います。

2)「外部参照」を参照（第 3.3.2 節 P.88）。

3)「トラブルシューティング」を参照（第 4.4 節 P.146）。

4)「属性」を参照（第 4.5.1 節 P.155）。

❖ リンクの順番

ライブラリからオブジェクトファイルをリンクする場合，**HLK** と LK とではリンクされる順序が異なります。したがって，**HLK** で作成された実行ファイルは，LK で作成された実行ファイルと比べてまったく同じ内容ではありません[5]。

オブジェクトファイルだけをリンクした場合は，リンクされる順序は同じですが，シンボルテーブル，シンボリックデバッグ情報 の部分は順序が異なります。つまりライブラリをリンクせず，`-x` オプションを与えてリンクしたときのみ，完全に実行ファイルの内容が一致します。

❖ 異なるタイプのライブラリの指定が可能

ライブラリ に，アーカイブ形式 (XC Ver. 1.x) とライブラリアン形式 (XC Ver. 2.x) の 2 つのタイプ[6]を混在させて指定することができます。混在させることによって，リンク時間に影響が出ることはありません。

❖ 相対セクション

相対セクション[7]は，SX–Window の `OBJR` 形式をサポートするために新しく作られたセクションです。相対セクションに対応するために，オブジェクトファイルのフォーマットが拡張されています。**HLK** では拡張されたフォーマットに対応しています。

❖ コモンエリアのシンボルの扱い

HLK では，コモンエリアのシンボル名と普通のシンボル名が衝突した場合，コモンエリアのシンボルとその領域を無効にしてリンクします。これがどのようなときに起こるかを，アセンブラの場合と C 言語の場合の 2 つの例を紹介します。

- **アセンブラの場合**

  List 4-2 と List 4-3 の 2 つのファイルをアセンブルしてリンクしようとした場合，List 4-2 の `.comm` 文は無効になり，List 4-3 の `.dc.b` 文の部分が有効になります。この場合 ワーニング が発生しますが，“`-w`” オプションの指定で抑制できます。

List 4-2 ● common.s

```
1:          .comm   symbol,10
```

List 4-3 ● symbol.s

```
1:          .xdef   symbol
2:
3:  symbol: .dc.b   1,2,3,4,5,6,7
```

5) 内容が異なるといっても，プログラムの動作が変わってしまうわけではありません。しかし，偶然にも正常に動いているように見えていたプログラムの場合，正常な動作をしないこともあります。今のところ，そのような報告はないようです。

6) XC Ver. 1.x のライブラリファイルの拡張子は ‘.a’，XC Ver. 2.x のライブラリファイルの拡張子は ‘.l’ になっています。

7) 「相対セクション」を参照(第 3.3.4 節 P.89)。

```
A>has common.s
A>has symbol.s
A>hlk common.o symbol.o
Warning, duplicate definition : symbol
A>
A>hlk -w common.o symbol.o
A>
```

- **C 言語の場合**

List 4-4 と List 4-5 をコンパイルしてリンクしようとした場合，List 4-4 の“int warning_symbol”の部分は無効になります。

List 4-4 ● common.c

```
1:  int warning_symbol;
```

List 4-5 ● symbol.c

```
1:  #include <stdio.h>
2:
3:  int warning_symbol = 20;
4:
5:  void main (int argc, char **argv)
6:  {
7:    printf ("warning_symbol = \%d\bs n", warning_symbol);
8:  }
```

```
A>gcc common.c symbol.c -o sample.x
Warning, duplicate definition : _warning_symbol
A>
A>sample
warning_symbol = 20
A>
```

当然ながら，コモンエリアのシンボル以外のシンボルが衝突した場合はエラーになります。

g++ (GNU C++ Compiler) でコンパイルしたファイルをリンクしようとすると，LK の仕様ではエラーとなってしまいますが，**HLK** ではg++に対応するために，エラーにしないように仕様を変更しました。ちなみに，**UNIX** のローダー[8]はエラーやワーニングなしでリンクしてくれるようです。

8)UNIX ではリンカのことをローダー (ld) と呼びます。

❖ 起動時のタイトル表示

HLK は起動しても，デフォルトで LK のようにタイトルを表示しません。LK のように起動時にタイトル表示をすることにより，リンク処理を始めたことを確認したい場合は，環境変数 SILK または silk に“-t”を追加してください。もし，指定できるならば **HLK** に直接このオプションを与えてもか

まいません。

❖ **テンポラリファイル**

HLK ではテンポラリファイル を作成しません。したがって，LK ではテンポラリファイルのディレクトリを設定するオプションであった “-t” は，起動時にタイトルを表示するオプションになりました。

❖ **Z ファイル**

Z ファイル は **HLK** ではサポートされていません。したがって，LK にあった -b オプションはありません。

4.4 ……………………… トラブルシューティング

リンカで発生するエラーは，その性格上，コンパイラやアセンブラで出力するエラーとは異なります。つまり，ソースファイルとオブジェクトファイルとの対応関係が失われてしまっているため，エラーの発生した箇所がわかりにくくなっているのです。しかし，発生するエラーと解決方法はかぎられています。このセクションでは，そのようなエラーの事例とその対処方法について解説します。

もし，この事例でも解決できない場合は，リンカのリンク手順を覚えることをお勧めします。アセンブラの知識も必要になってしまいますが，エラーが起きても困ることはないでしょう。

4.4.1 Undefined symbol(s) in *XXXX* エラー

リンク時に出るエラーはほとんど Undefined symbol(s) といってもよいでしょう。最初はとまどいますが，慣れてくれば簡単に解決できるようになります。解決方法は，次の3つのうちのどれかに当てはまります。

1. 表示されたシンボルの文字がタイプミスによるものか調べる

C 言語のプログラムの場合，jugemjugem () を呼ぶところをまちがって，jugemjugemu () とタイプしてしまうと「_jugemjugemu なんてシンボルはない」と怒られてしまいます。ここで，関数名 jugemjugemu の前に“_”がついているのに気がついたと思います。これはコンパイラが，グローバルな関数名や変数名を表すシンボル名の前に“_”をつけるようになっているためです[1)]。

初めてこのエラーに遭遇すると,「なぜコンパイルはエラーなしでできたのにリンカでエラーが発生してしまうのだろう」と悩んでしまいそうです。しかし，このエラーが関数名をタイプミスしたことで，同名の関数をリンカが見つけられなかったということに気がつけば，解決方法は….。そうです，ソースファイル中の該当する関数名を修正してあげればよいのです。修正すべきソースファイル名は，エラーが発生したときのオブジェクトファイルの名前からわかります。

逆に，呼び出される側の関数名をまちがえたりする場合もあるので，そちらのほうもチェックしてみましょう。ほかにも，グローバル変数名をまちがったりすると同じようなエラーが出ますが，解決方法は基本的に同じです。

1)余談ですが，C++ ではさらに関数の引数や戻り値の情報もシンボル名につけ加えます。従来の C 言語のライブラリを使用できるように，C 言語と同じようなシンボル名を使うこともできるようになっています。

2. ライブラリを指定するのを忘れていないか

C 言語のプログラムの場合，FILES () や G_CLR_ON () などのように DOS または IOCS ライブラリにある関数を呼んだのに，呼ばれた関数が入っているライブラリを指定しないと，このエラーが発生します。この場合は，_FILES や _G_CLR_ON が未定義外部参照シンボルとなります。

この場合は，DOS と IOCS のライブラリがなかったので，**doslib.a** と **iocslib.a** をリンクしてあげればよいわけです[2]。**GCC** を使用してリンクする場合は，“**-ldos -liocs**” を指定すると，**doslib.a** と **iocslib.a** がリンクされます。

2)XC Ver. 2.x の場合は，doslib.l と iocslib.l です。

XC Ver. 2.x のライブラリを使用していて **GCC** を使用しているときは，必ず **floateml.l** か **floatfunc.l** をリンクするようにしてください。**GCC** を使用してリンクする場合は，**-lfloateml** または **-lfloatfunc** を指定します。

3. シンボル名が外部宣言されているか

C 言語の場合，関数を定義したときに **static** 宣言してあるのに，他のファイルでその関数を呼んでいる場合があります。**static** 宣言をはずすなり，関数の定義を呼んでいるソースへ移動するなりして解決しましょう。

アセンブラの場合は，**.xdef** か **.global** で外部宣言をするのを忘れていたりする場合に起こります。

4.4.2 Duplicate definition エラー

このエラーを取り除くのは，それほど難しくはありません。これはエラーメッセージで出たファイルを調べれば，ほとんど解決します。しかし，難しい場合も例外的にあります。

◆ 簡単なケース

たとえば，**dup1.o** と **dup2.o** をリンクしたときに，このエラーが発生したとします。この場合，**dup1.o** か **dup2.o** のどちらかのエラーが発生したシンボル名 (または関数名) を削除するなり，変更することで解決できます。

C 言語の場合，複数のファイルで同一のグローバル変数を初期化しようとした場合に，このエラーが発生することがあります。この場合は，初期化する部分を 1 つ残して，他の部分を extern 宣言してください。簡単なサンプルを紹介します。

今，次の 2 つのプログラム List 4-6 と List 4-7 をコンパイル／リンクしたとします。

List 4-6 • init1.c

```
1:  extern int global_var  = 1;
2:  extern int global_var2 = 2;
3:
4:  void main (int argc, char **argv)
5:  {
6:    printf ("g_var = %d, g_var2 = %d\n", global_var, global_var2);
```

```
7:  }
```

List 4-7● init2.c

```
1:  int global_var  = 1;
2:  int global_var2 = 2;
```

```
A>gcc init1.c init2.c
Duplicate definition : _global_var
 in init1.o init2.o
Duplicate definition : _global_var2
 in init1.o init2.o

H:/lang/gcc/gcc.x: Program hlk.x exit status 65535.
```

List 4-6とList 4-7を見ると，両方でグローバル変数gloval_varとglobal_var2を初期化しているようです。そこで，List 4-6を修正してみます。

List 4-8● 修正後のinit1.c

```
1:  extern int global_var;   /* 変数を参照するように変更 */
2:  extern int global_var2;  /* 変数を参照するように変更 */
3:
4:  void main (int argc, char **argv)
5:  {
6:    printf ("g_var = %d, g_var2 = %d\n", global_var, global_var2);
7:  }
```

List 4-8はグローバル変数を参照するように，List 4-7ではグローバル変数を定義するようになりましたので，シンボルの衝突が解消されたはずです。コンパイル，リンクしてみると，次のように，無事エラーがなくなりました。

```
A>gcc init1.c init2.c
A>
```

◆ 複雑なケース

複雑な場合というのは，シンボルとオブジェクトファイルの依存関係が入り組んでいることがおもな原因ですので[3]，以下の文章は関係をよく把握しながら読んでください。

3)この原稿を，書いている本人がややこしくしているからという話もありますが....。

最初，prog.oとlibraries.aで実行ファイルを作成していたのですが，どうしてもライブラリの仕様を変更しなければならなくなってしまいました。そのためlibraries.aの_main.oを改造して，my_main.oを作成します。これをリンクしてみると，

```
A>dir
 ****                   A:\
    3 ファイル         **K Byte 使用中      ****K Byte 使用可能
 ファイル使用量        **K Byte 使用
my__main          o          ***  **-**-**  **:**:**
prog              o          ***  **-**-**  **:**:**
libraries         a          ***  **-**-**  **:**:**
A>
A>hlk prog.o libraries.a
A>
A>hlk prog.o my__main.o libraries.a
Duplicate definition : __main
 in my__main.o __main.o
Duplicate definition : _label1
 in my__main.o __main.o
Duplicate definition : _label2
 in my__main.o __main.o
A>
```

なぜか，リンクするつもりがない libraries.a の__main.o がリンクされてしまっています。prog.s[4] と my__main.s[5] を見ても，__main.s を参照するようなシンボルは使用していないようです。

4)prog.o のソースファイル (List 4-9)。

5)my__main.o のソースファイル (List 4-10)。

List 4-9 ● prog.s

```
 1:          .xref    __main
 2:
 3:  _main:
 4:          .dc.w    $ff00
```

List 4-10 ● my__main.s

```
 1:          .xref    _subroutine
 2:          .xdef    __main
 3:          .xdef    _label1,_label2,_label3
 4:
 5:  __main:
 6:          jsr      _subroutine
 7:
 8:          *
 9:          * 自分で追加したプログラム
10:          *
11:
12:          rts
13:
14:  _label1:
15:  _label2:
16:  _label3:
17:
18:          .end     __main
```

List 4-11 ● __main.s

```
 1:          .xref    _subroutine
 2:          .xdef    __main
 3:          .xdef    _subroutine_ref_point
 4:          .xdef    _label1,_label2,_label3
 5:
 6: __main:
 7:          jsr      _subroutine
 8:          rts
 9:
10: _subroutine_ref_point:
11:          ds.l     1
12:
13: _label1:
14: _label2:
15: _label3:
16:
17:          .end     __main
```

my__main.s と **__main.s** を見比べてみると，**my__main.s** には，外部定義シンボル **_subroutine_ref_point** が定義されていません。どうやら，これが原因のようです。調べてみると，このシンボルは **libraries.a** 中の **subroutine.o** にあるサブルーチン **_subroutine** が参照していました。

もし，このシンボルが **my__main.s** で外部定義されていなかったことがエラーの原因ならば，このシンボルを外部定義するように **my__main.s** を変更した，**my__main.o** (List 4-12) を使用してリンクしてみれば，エラーは発生しなくなるはずです。

List 4-12 ● 修正した my__main.s

```
 1:          .xref    _subroutine
 2:          .xdef    __main
 3:          .xdef    _label1,_label2,_label3
 4:
 5: __main:
 6:          jsr      _subroutine
 7:
 8: *
 9: * 自分で追加したプログラム
10: *
11:
12:          rts
13:
14: _subroutine_ref_point:
15:          ds.l     1
16:
17: _label1:
18: _label2:
19: _label3:
20:
21:          .end     __main
```

```
A>hlk prog.o my__main.o libraries.a
A>
```

実際に _subroutine_ref_point を外部定義するようにすると，エラーは出なくなりました。せっかくエラーが出ないようにできたのですから，なぜエラーが発生したのか，エラーが発生するまでのリンカの流れを追ってみましょう[6]。

6)リンカの動作の詳細は，次節「リンカの動作」を参照してください(P.155)。

1. **prog.o と my__main.o をリンクする**
2. **外部参照シンボル _subroutine がないので，libraries.a の subroutine.o をリンクする**
3. **subroutine.o では _subroutine_ref_point を外部参照しているが，定義されていないので libraries.a の __main.o がリンクされる**
4. **__main.o で外部定義されているシンボルは，すでにリンクされている my__main.o で外部定義されているため，Duplicate definision エラーが発生する**

この場合のエラーは，変更した my__main.o には，直接的な原因がなく，間接的なものだったので見つけにくいものになっていたようです。

シンボルの参照を調べるには，マップファイルを出力して，それを見ると参考になります。

4.4.3 Relative error, Over flow エラー

どちらも，シンボルの値を PC 相対アドレス形式[7]で外部参照しようとしたときに発生するので，解決方法は同じになります。

7)PC(プログラムカウンタ)からの相対位置でアドレスを表す方法。

LK ではエラーメッセージからわかることは，エラーが起きたオブジェクトファイル名だけでしたので，実際にオブジェクトファイルのどの位置でエラーが発生したのか，見つけ出すのは大変でした。**HLK** ではエラーが起きた箇所を発見しやすいように，少し詳しくレポートするようにしてあります。

◆ アセンブラの場合

それでは，次の 2 つのプログラム List 4-13 と List 4-14 を例にして説明します。

List 4-13 ● far_symbol.s

```
1:          .xdef   far_symbol,far_entry
2:
3:          .ds.b   65536
4:  far_symbol:
5:  far_entry:
6:          rts
7:          .ds.b   65536
8:          .end
```

List 4-14 ● far_ref.s

```
 1:          .xref    far_symbol,far_entry
 2:
 3:  start:
 4:          move.w   #10,d0
 5:          lea      far_symbol(pc),a0       * エラーが起こる部分
 6:  loop:
 7:          move.w   d0,(a0)+
 8:          dbra     d0,loop
 9:
10:          bsr      far_entry               * エラーが起こる部分
11:
12:          .dc.w    $ff00
13:
14:          .end     start
```

この 2 つのファイルをアセンブル／リンクすると，当然ながらエラーが出ます。この場合はプログラムの規模が小さいので，List 4-14 の 5 行目と，10 行目でエラーが発生することは簡単にわかりますが，ファイルが大きくなると，このエラーは発見しにくくなります。

```
A>has far_symbol.s
A>has far_ref.s
A>
A>hlk far_symbol.o far_ref.o
Over flow in far_ref.o
 at 00000006 (text)
Over flow in far_ref.o
 at 00000010 (text)
A>
```

エラーの at *XXXX* の部分に注目すると，このエラーメッセージは，**far_ref.o** の **text** セクションのアドレス (16 進数) 00000006 と 00000010 でエラーが発生したことを示しています。これを，手がかりにしてエラーの原因箇所を突き止めてみます。

まず，エラーが発生した **far_ref.s** を再アセンブルします。このとき，**-p** オプションを指定してリストファイルを作成します。リストファイルにはソースファイルのある行がどのアドレスに対応しているかの情報が出力されるので，このファイルを調べることによって，ソースファイルの何行目でエラーが発生したかが特定できます。

```
A>has far_ref.s -p
A>
A>type far_ref.prn
         :
         :
<far_ref.s>
    1 00000000                           .xref   far_symbol,far_entry
    2 00000000
    3 00000000                   start:
    4 00000000 303C000A                  move.w  #10,d0
    5 00000004 41FA????                  lea     far_symbol(pc),a0
    6 00000008                   loop:
    7 00000008 30C0                      move.w  d0,(a0)+
    8 0000000A 51C8FFFC_00000008         dbra    d0,loop
    9 0000000E
   10 0000000E 6100????                  bsr     far_entry
   11 00000012
   12 00000012 FF00                      .dc.w   $ff00
   13 00000014
   14 00000014                           .end    start
         :
         :
A>
```

far_ref.prn で，テキストセクションのアドレスが 00000006 と 00000010 を含んでいる部分を捜すと ...，5 行目と 10 行目だということがわかります。両方とも，PC 相対アドレス形式で，± 32K の範囲を超えた場所をサブルーチンコールしているためにエラーが発生しています。

PC 相対アドレス形式で届かなかったため，絶対アドレス形式[8]でサブルーチンコールするように 5 行目と 10 行目を修正すると，List 4-15 のようになります。

8)PC やレジスタの値に関係なく，つねに特定のアドレスを表す方法です。

List 4-15 ● 修正した far_ref.s

```
 1:          .xref   far_symbol,far_entry
 2:
 3:  start:
 4:          move.w  #10,d0
 5:          lea     far_symbol,a0    * 絶対アドレッシング
 6:  loop:
 7:          move.w  d0,(a0)+
 8:          dbra    d0,loop
 9:
10:          jsr     far_entry        * 絶対アドレッシング
11:
12:          .dc.w   $ff00
13:
14:          .end    start
```

修正したファイル List 4-15 をアセンブル／リンクしてみます。

```
A>has far_ref.s
A>
A>hlk far_symbol.o far_ref.o
A>
```

上記のように，エラーはなくなりました。

◆ C 言語の場合

アセンブリ言語についての知識がないと対処は難しいのですが，**GCC** を使っている場合はとりあえず **-fall-jsr** オプションと **-fstrings-nopcr** を使用してみてください。SX-Window の OBJR 形式のプログラムの場合は，さらに **-fall-remote** もつけてみてください。

アセンブリ言語についての知識がある場合は，**GCC** に -S オプションをつけてコンパイルして，アセンブラソースファイルを出力します。後は，アセンブラの場合[9]と同じような手順で対処します。

9)第 4.4.3 節 P.151 参照。

4.5 ……………………………… リンカの動作

このセクションでは，オブジェクトファイルから実行ファイルができるまでを説明します。この部分は，特に知らなくてもリンカを使用できるので読み飛ばしてもかまいませんが，今までブラックボックス的にしか見えなかったリンカの動作を理解するのに役立つと思います。

4.5.1 用語の説明

まずはじめに，以下の説明でよく出てくる用語の意味を明確に定義することにしましょう。

❖ **シンボル**

俗にラベルとも呼ばれています。シンボルには，自分の値をもっている「外部定義シンボル」と，値を参照している「外部参照シンボル」の 2 種類があります。

❖ **属性**

このセクションでは，シンボルの属性を指します。これはシンボルがもっている値がどのような意味をもっているかを表しており,「絶対値」と「アドレス」に大きく分かれます。絶対値には数値が入り，アドレスはプログラムの位置を指しています。アドレスのシンボルは，絶対値のシンボルと次の 2 つの点が異なります。

- **書き込まれたシンボル値は，プログラムが実行されるアドレスにより変更される**
- **属性がアドレスのシンボルどうしの演算は引算しか許されない**

また，アドレスの属性はそれがどのセクションの値かという情報ももっています。値はセクションの先頭からの位置を示します。

❖ **リンクされる**

このセクションでは一般的なリンクと異なり，実行ファイルを作成するためのオブジェクトファイルとして登録されたことを示しています。

❖ **相対オフセットテーブル**

HAS.X で新しくできたセクションである **rdata**，**rldata** に書き込まれる相対セクションの値は，プログラムを自分でリロケートしなければいけませ

ん。このテーブルは，書き込まれた相対セクションの値をリロケートするための情報のテーブルです。

それでは，大まかな全体の流れを見てみましょう。リンカの処理の手順をFig. 4-1に示します。

ファイルの読み込み
⇓
外部参照シンボルの解決とリンク
⇓
シンボルの値の決定
⇓
実行ファイルの作成

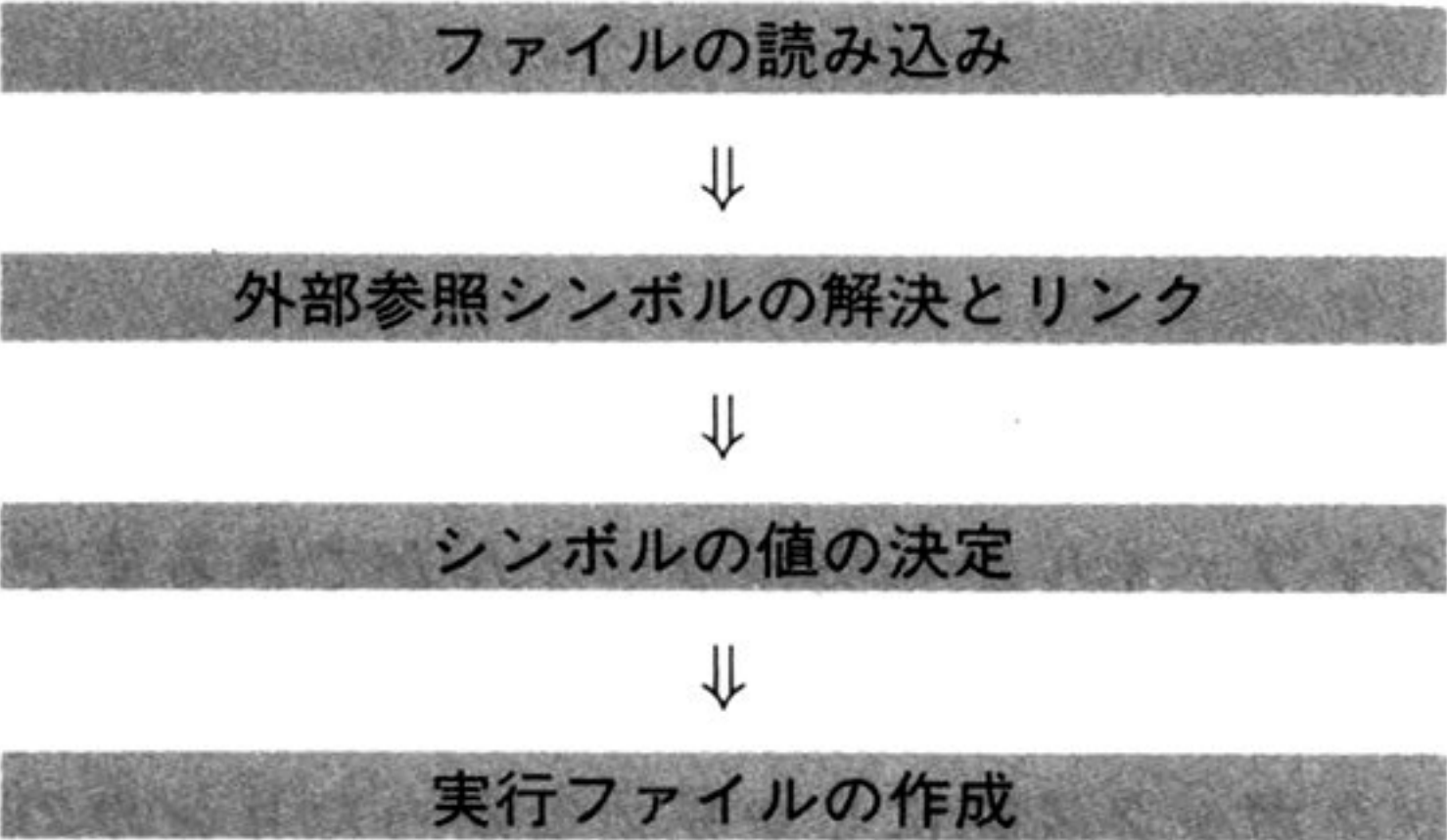



Fig. 4-1● リンカの処理の流れ

4.5.2 リンカの動作の詳細

大まかな流れを簡単に紹介したところで，リンカの動作を細かく見ていきます。

1. コマンドラインの解析

まず，与えられたコマンドラインから，オプションを抜き出し処理します。残った部分をファイル名とみなします。したがって，先頭の文字が“-”(ハイフン)でなければファイル名とみなします。

2. ファイルの読み込み

次に指定されたオブジェクト，ライブラリファイルを読み込みます。読み込んだファイルは先頭の数バイトを見て，どのタイプかを判断して，それぞれ処理します。もし，オブジェクトファイルでもライブラリファイルでもない場合は，そのファイルを無視して処理を進めます。Table 4-2に，ファイルの内容とそのタイプの対応を示します。

Table 4-2● ファイルの内容とファイルタイプ

ファイルの内容 (16進数)	ファイルのタイプ
d0 00	オブジェクトファイル
d1 00 00 00 00 02	ライブラリファイル (アーカイブ形式)
00 68	ライブラリファイル (ライブラリアン形式)

このように，オブジェクトファイルとライブラリファイルの処理はほとんど同じなのですが，オブジェクトファイルの場合は内容にかかわらず，すぐにリンクされます。またライブラリファイルの場合は，複数のオブジェクトファイルに分解されて処理され，この時点ではリンクされません。

3. シンボルテーブルの作成

読み込まれたオブジェクト(またはライブラリ)ファイルは，すべてシンボル

テーブルが作成されます。シンボルテーブルは2つからなり，1つは外部参照シンボルのテーブル，もう1つは外部定義シンボルのテーブルです。コモンエリアのシンボルは，外部定義シンボルと外部参照シンボル両方の性質をもっているように扱われます。

また外部定義シンボルにはシンボル名から，そのシンボルを定義しているオブジェクトファイルがわかるような外部定義テーブル[1)]があるので，そこへ登録されます。ここで，オブジェクトファイルとライブラリファイルで同じ名前の外部定義シンボルが定義されている場合は，登録順にかかわらずオブジェクトファイルの外部定義シンボルが優先されます[2)]。その結果，ライブラリに入っているC言語の関数をユーザプログラムがオーバーライドできます[3)]。

そして，アドレスを表す属性の外部定義シンボルの値は「定義元のオブジェクトファイルの**セクションの先頭から何バイト目を指している」のように表されています。

4. 外部参照シンボルの解決

さらにリンクされたオブジェクトファイルは，自分が所有している外部定義シンボルが有効であり，自分のものだということを外部定義テーブルに書き込みます。このとき，別のオブジェクトファイルによって，すでに外部定義シンボルが所有されている場合はエラーになります。ただしコモンエリアどうしの場合はエラーにならず，サイズが大きいコモンエリアのオブエジェクトファイルが所有します。

また，リンクされた各オブジェクトファイルに外部参照シンボルが存在した場合は，同じ名前の外部定義シンボルを捜します[4)]。もし，そのシンボルを所有しているオブジェクトファイルがまだリンクされていない場合はリンクします。この処理を最後に，リンクされたオブジェクトファイルまで繰り返します。

5. 各セクション，テーブルの大きさの決定

上記のように，リンクされるオブジェクトファイルが決定したら，各オブジェクトファイルが使用している相対オフセットテーブルの大きさとコモンエリアの大きさを計算します。

コモンエリアの大きさを求めるには，リンクされたオブジェクトファイルの外部定義シンボルテーブルを調べ，シンボルがコモンエリアの属性ならばそのサイズを加算します。ただし単純に加算すると，同名のコモンエリアのシンボルが複数あるため正しく計算されません。そのため外部定義テーブルを参照して，自分が所有しているシンボルの場合のみ加算するようになっています。

また相対オフセットテーブルには，各オブジェクトファイルのrdata，rldataセクションで相対セクション(rdata ~ rlstack)の属性の値を書き込んでいる位置を調べて，その位置をテーブルに登録します。

さて，コモンエリアの大きさがリンク時点で決定できないのは，コモンエリアの性質のためです。コモンエリアは，同じ名前のコモンエリアが見つかっ

1)これ以降，外部定義テーブルはこのテーブルを指すことにします。

2)すでにオブジェクトファイルの外部定義シンボルが登録されている場合は何もしません。

3)つまり，C言語のライブラリ関数であるprintf()を自分のプログラムで定義すると，ライブラリのprintf()ではなくて自分のプログラムのprintf()が使用されます。このため，まちがってread()関数を自分で定義してしまうと思わぬ誤動作に悩まされてしまいます(read()関数は他の関数に使われている!)。これは，実際によくある話だったりします。

4)シンボルを捜す条件は名前だけで，属性を指定することはできません。シンボルの属性を細かく定義できて，リンク時にそれをチェックできる機構があれば，分割コンパイル／リンク時の変数の型チェックも可能になるのですが....。

た場合は，その領域の大きさを調べて，より大きいほうの領域を確保しなければいけません。つまり，すべてのオブジェクトファイルがリンクされないと，コモンエリアの大きさが決定できないわけです。

このように，相対オフセットテーブルの大きさがリンクするまで決定できないのは，List 4-16，List 4-17 のような場合があるからです。

これらは，外部参照しているシンボル (または変数) `hook_data` の値を `rdata` (または `rldata`) セクションへ書き込んでいます。外部参照シンボルは，同名の外部定義シンボルが見つかるまで属性が決定できないので，この部分がテーブルに登録されるべきかどうかをリンク時に決定できないわけです。

List 4-16 ● すぐに決定できない例 (C 言語の場合)

```
1:  extern int hook_data;
2:  int *hook_datap = &hook_data;
```

List 4-17 ● すぐに決定できない例 (アセンブラの場合)

```
1:          .xref   data_table1,data_table2
2:
3:          .rdata
4:
5:  table_list:
6:          .dc.l   data_table1,data_table2
```

6. 実行ファイルの作成

以上でリンクされるオブジェクトファイル，外部参照シンボルの解決，各セクション，テーブルの大きさが決定したので，実行ファイルを作成する条件は整いました。

まず，各セクションの大きさがわかったので，属性がアドレスであるシンボルの実行ファイルの先頭からの値を決定します。

次にリンカはリンクされたオブジェクトファイルのコマンドを解釈して実行します。オブジェクトファイルのコマンドは大別すると，定義 (シンボルの定義や各セクションのサイズ情報)，制御 (セクションの切り替え／終了)，演算 (シンボルや定数の演算)，書き込み (シンボルの値や定数の書き込み) の 4 つに分けられます。実際に機能するのは，定義以外の 3 つのコマンド群です。もし演算や書き込みを行ったときにエラーが発生した場合は，それらのエラーに関する情報を表示します。たとえば，演算時にオーバーフローを起こしたり，書き込み時に書き込める値の範囲を超えてしまった場合などです。オブジェクトファイルのどの位置を実行しているかがわかっているので，それらの情報を表示します。

ロングワードの値を書き込んだときに，その値の属性がアドレスの場合はリロケートテーブルに登録されます。リロケートテーブルには，前回登録したときのアドレスと現在のアドレスの値の差を登録します。差と実際に登録されるコードは，Table 4-3 のとおりです。

そして，すべてのオブジェクトファイルのコマンドの実行が終了した時点で

実行ファイルの完成です。後は，シンボル情報と シンボリックデバッグ情報をつけるだけです。
シンボル情報とシンボリックデバッグ情報のフォーマットについては，251ページを参照してください。

Table 4-3● 差の値と登録されるコードの関係

差の値の範囲（x）	登録されるコード	
$00000000 ～ $0000ffff	x の下位ワード	(2 バイト)
$00010000 ～ $ffffffff	$0001 x	(6 バイト)

Chapter 5

GDB

GDB は，FSF で開発されたソースレベルデバッガです。この開発キットの GDB は，FSF の GDB を Human68k に移植したものであり，GDB 本来の機能に加え，独自の改良を施したデバッガです。本章では，GDB のコマンドについて詳しく説明します。

5.1 …………………………………… GDB

GDB の説明に入る前に，まずデバッガというプログラム開発ツールについて簡単に説明します。次に，**GDB** の特徴について説明します。

5.1.1 デバッガ

デバッガは，プログラマの制御下でプログラムを実行することにより，バグや不具合などを調査していくためのツールです。プログラムの開発過程で多くの時間を必要とするのはデバッグ作業ですから，デバッガを利用することでデバッグ作業を効率よく行うことができ，プログラムの開発期間を短縮することが可能です。

デバッガには，マシンレベルデバッガ[1]とソースレベルデバッガ[2]があります。前者はマシン語[3]レベルでのデバッグを目的としていますが，大きなプログラムや複雑なプログラムのデバッグには大変な労力を必要とします。さらにマシンのアーキテクチャやマシンの命令も理解している必要もあります。後者は C 言語などの高級言語で書かれたプログラムのデバッグを目的とし，ソースプログラムを 1 行単位で実行させることにより，状態の変化を調査していくことが容易にできます。

1)XC の db.x などです。

2)gdb.x や XC の scd.x などです。

3)アセンブラなどです。

5.1.2 GDB

GDB は，FSF で開発されたソースレベルデバッガで，豊富で強力な機能をもっています。デバッグ可能な言語は，C，C++[4]で，それぞれの言語に最適なデバッグ環境を提供します。

Human68k 版 **GDB** は，通常の C 言語で開発されたアプリケーションのデバッグ機能に加え，SX–Window アプリケーション開発のために本書の **X68000 GCC** で拡張された記憶域クラスに対応しています。また，デバッガの画面とアプリケーションプログラムの画面を切り替える機能を 1 台のモニタで実現するデバッグ環境もサポートしています。

4)g++(GNU C++) に対応しています。

5.1.3 GDB の機能

GDB は，次のような機能およびそれらをサポートする他の機能をもっています。

- **ステップ実行**[5]
  プログラムを 1 行単位または複数行実行させて停止させる
- **プログラムの動作を詳しく調べる**[6]
  変数の値の表示，値の変更または配列や構造体のような複雑なデータ構造の値でも表示，変更することができる
- **プログラムの実行による変数の値の変化を監視する**[7]
- **プログラムを実行し，その実行結果が及ぼすであろうさまざまなことを報告する**[8]
- **指定された状況において，プログラムの実行を停止する**[9]
- **何が起こっているのかを検査する**[10]
  そのことによってプログラムが停止したとき，バグの原因を知ることができる

5)第 5.4.4 節 (P.190) 参照。

6)第 5.6 節 (P.197) 参照。

7)第 5.6.3 節 (P.202) 参照。

8)第 5.5 節 (P.193) 参照。

9)第 5.7 節 (P.207) 参照。

10)第 5.8 節 (P.214) 参照。

5.1.4 GDB の特徴

また，GDB には次のような特徴があります。

- **使いやすい対話型ユーザインタフェイス**[11]
- **デバッグしているプログラム中の関数の呼び出し能力**[12]
- **変数値の履歴**[13]
- **ユーザが定義できるコマンド**[14]

11)第 5.3.6 節 (P.186) を参照。

12)第 5.6.6 節 (P.205) を参照。

13)第 5.6.4 節 (P.204) を参照。

14)第 5.12.1 節 (P.224) を参照。

◆ ユーザインタフェイス

GDB は，コマンドによる対話型ユーザインタフェイスを採用しています。つまり，GDB がプロンプト[15]を表示し，コマンドが入力可能であることを示している状態で，コマンド名とそのコマンドに必要な引数を，続けて 1 行で入力します[16]。入力中は，テキストエディタのような行編集が可能です。また，コマンドまたはデバッグしているプログラムの中の関数名／変数名にも有効なコンプリーション機能[17]，ヒストリ機能[18]などを使いこなせば，最近の GUI[19]を使ったデバッガにも劣らないデバッグ環境になります。

15)通常は '(gdb)' です。

16)「コマンド入力」を参照（第 5.3.6 節 P.186)。

17)「コンプリーション機能」を参照（第 5.3.6 節 P.186)。

18)「ヒストリ機能」を参照（第 5.3.6 節 P.186)。

19)マウスを使って命令します。

◆ プログラム中の関数の呼び出し

変数の内容を表示するのと同じ感覚で，デバッグしているプログラムの中の関数を直接実行し，テストすることができます[20]。用途としては，デバッグ対象プログラムの中にプログラムの状態を詳しく表示するような関数[21]をあらかじめ作っておき，デバッグ時にそれを呼び出して調査することができます。また，関数を引数の値を変えてテストする場合に有効な機能でしょう。

20)第5.6.6節(P.205)参照。

21)サブルーチン。

◆ 変数値の履歴

変数の内容を表示するためのコマンドとして，“print”コマンドがあります。この“print”コマンドによって表示された値は，他の式から参照することができるように**GDB**の中に保存されます[22]。

22)第5.6.4節(P.204)参照。

◆ ユーザ定義コマンド

GDBが標準で用意しているコマンドを組み合わせることによって，新たにコマンドを作ることができます。コマンドには，他のコマンド同様，“help”コマンド[23]で表示されるドキュメントをつけることもできます[24]。

23)コマンドの説明を表示します。

24)第5.12.1節(P.224)参照。

5.2 ………… とりあえず GDB を使ってみる

GDB を使うのが始めてというユーザのために，基本的な機能を SX–Window 上で動作するサンプルプログラム List 5-1 を使って，次のように簡単に説明していきます。すでに，GDB を使ったことのあるユーザは読み飛ばしてもかまいません。

- **GDB を起動する**[1]
- **ブレークポイントを設定する**[2]
- **デバッグ対象プログラムの実行を開始する**[3]
- **変数の値を表示または変更する**[4]
- **デバッグを終了する**[5]

1)「GDB の起動」を参照（第 5.3 節 P.184）。

2)「ブレークポイント」を参照（第 5.7 節 P.207）。

3)「プログラムの実行」を参照（第 5.4 節 P.189）。

4)「データを調べる」を参照（第 5.6 節 P.197）。

5)「GDB を終了する」を参照（第 5.3 節 P.184）。

5.2.1 デバッグの準備

まず，サンプルプログラムについて簡単に解説した後，プログラムのコンパイルについて説明します。

◆ サンプルプログラム

List 5-1 は，算術代入文を読み込んで逆ポーランド記法に変換するプログラムです。このプログラムは GDB の基本的な使い方を説明するために使用するもので，実用を目的としたものではありません。

まず付録ディスクから“`test.c`”(List 5-1) を，デバッグ作業を行うワーキングディレクトリにコピーしてください。またカレントディレクトリは，`test.c` をコピーしたディレクトリに設定しておいてください。

逆ポーランド記法

逆ポーランド記法による算術式は，日本語の文法に似ています。たとえばA+Bは「AとBを足す」という語順のとおりにAB+と表し，AB+C*は「AとBを加え，それにCをかける」という意味です。また逆ポーランド記法は，演算の優先順位を指定するカッコを必要としません。

例題

算術代入文	逆ポーランド記法
A+B	AB+
A*B-C/D	AB*CD/-
A*(B+C)-D	ABC+*D-

List 5-1 ● test.c

```
 1:  /*******************************************************************/
 2:  /*                                                                 */
 3:  /*      Polish notation                                            */
 4:  /*                                                                 */
 5:  /*******************************************************************/
 6:
 7:  #include <stdio.h>
 8:  #include <sys/stat.h>
 9:  #include <sxlib.h>
10:
11:  #define BUFFMAX 20                          /* 最大文字数 */
12:
13:  void nullEvent (void);
14:  void mouseLDownEvent (void);
15:  void mouseRDownEvent (void);
16:  void keyDownEvent (void);
17:  void updateEvent (void);
18:  void activateEvent (void);
19:  void app12Event (void);
20:  int Init (void);
21:  void EndPr (int, long);
22:  void polish (void);
23:  void display (void);
24:
25:  tsevent eventrec;                   /* イベントレコード */
26:  int taskid;                         /* 自分のタスク ID */
27:  int sxver;                          /* SX システムのバージョン */
28:  int eventmask  = 0x3282;            /* イベントマスク */
29:  int initflag   = 0;                 /* 初期化フラグ */
30:  int activflag  = 0;                 /* アクティブフラグ */
31:  window *windowptr;                  /* <window> へのポインタ */
32:  tEdit **teHdl;                      /* <tEdit> へのハンドル */
33:  rect wsize;                         /* ウィンドウサイズ */
34:  char inbuf[21];
35:  char outbuf[21] = "";
36:  char *inbufptr;
37:  char *outbufptr;
38:  int ch;
39:  char wtitle[] = "\017Polish notation";   /* ウィンドウタイトル */
40:  rect terect = {16+48+16, 16, 240, 32};   /* <tEdit> の dest */
41:  common char _sxkernelcomm[] = "sxwdb.x -D -K -L0";
42:                                      /* カーネル起動コマンド */
43:  int main ()
44:  {
```

```
45:     int ret;
46:
47:     if ((ret = Init ()) < 0)
48:       EndPr (ret, 0);      /* 初期化に失敗したので終了する */
49:     while (1)
50:       {                    /* タスクマンのイベントを得る   */
51:         TSEventAvail (eventmask, &eventrec);
52:         switch (eventrec.what) {
53:           case E_IDLE:
54:             nullEvent ();
55:             break;
56:           case E_MSLDOWN:
57:             mouseLDownEvent ();
58:             break;
59:           case E_MSRDOWN:
60:             mouseRDownEvent ();
61:             break;
62:           case E_KEYDOWN:
63:             keyDownEvent ();
64:             break;
65:           case E_UPDATE:
66:             updateEvent ();
67:             break;
68:           case E_ACTIVATE:
69:             activateEvent ();
70:             break;
71:           case E_SYSTEM1:
72:           case E_SYSTEM2:
73:             app12Event();
74:             break;
75:         }
76:       }
77:   }
78:
79:   /*
80:   ** ヌルイベント
81:   */
82:
83:   void nullEvent (void)
84:   {                                   /* <graph> をカレントにセットする */
85:     GMSetGraph ((graph *)(windowptr));
86:     TMEvent (teHdl, (event *)(&eventrec)); /* カーソルをブリンクさせる */
87:   }
88:
89:   /*
90:   ** マウスレフトダウンイベント
91:   */
92:
93:   void mouseLDownEvent (void)
94:   {
95:     int ret;
96:     int reta;
97:                                       /* 自分のウィンドウ上かを調べる */
98:     if (windowptr == (window *)(eventrec.whom))
99:       {
100:                                      /* マウスダウンイベントを取除く */
101:        TSGetEvent (eventmask, &eventrec);
102:        if (activflag == 0)
103:          {
104:            WMSelect (windowptr); /* ウィンドウをアクティブに */
105:            ret = WMFind (*(point_t *)(&eventrec.whom2), (window **)(&reta));
106:            if (ret != W_INDRAG)
107:              return;            /* タイトルバー以外のときはリターン */
108:            if (EMLStill () == 0)
```

```
109:             return;              /* マウスが離されていたらリターン */
110:         }
111:         ret = SXCallWindM (windowptr, &eventrec);
112:         switch (ret)
113:           {
114:             case W_ININSIDE:     /* マウスがウィンドウの内側にある */
115:               GMSetGraph ((graph *)(windowptr));
116:               if (GMPtInRect (&terect, EMMSLoc ()) != 0)
117:                 TMEvent (teHdl, (event *)(&eventrec));
118:               else
119:                 ret = SXCallCtrlM (windowptr, &eventrec, 0, 0, 0);
120:               break;
121:             case W_INCLOSE:      /* クローズボタンが押された */
122:               EndPr (0, 0);      /* 終了 */
123:               break;
124:           }
125:       }
126: }
127:
128: /*
129: ** マウスライトダウンイベント
130: */
131:
132: void mouseRDownEvent (void)
133: {                                 /* 自分のウィンドウ上かを調べる */
134:   if (windowptr == (window *)(eventrec.whom))
135:     {
136:       TSGetEvent (eventmask, &eventrec); /* イベントを取り除く */
137:                                 /* <graph> をカレントにセットする */
138:       GMSetGraph ((graph *)(windowptr));
139:                                 /* ポップアップメニュー処理 */
140:       TMEvent (teHdl, (event *)(&eventrec));
141:     }
142: }
143:
144: /*
145: ** キーダウンイベント
146: */
147:
148: void keyDownEvent (void)
149: {
150:   TSGetEvent (eventmask, &eventrec);      /* イベントを取り除く */
151:   if ((eventrec.whom & 0xffff) == 0x0d)
152:     {
153:       int ret;
154:       ret = TMGetText (teHdl, inbuf, BUFFMAX);
155:       if (ret > 0)
156:         {
157:           inbuf[ret] = '\0';
158:           polish ();
159:           GMSetGraph ((graph *)(windowptr));
160:           display ();
161:         }
162:     }
163:   else
164:     {                           /* それ以外は 1 文字入力する */
165:       GMSetGraph ((graph *)(windowptr));
166:       TMEvent (teHdl, (event *)(&eventrec));
167:     }
168: }
169:
170: /*
171: ** アップデートイベント
```

```
172:  ** WMUpdate () をコールする前に必ずカーソルを消すこと
173:  */
174:
175:  void updateEvent (void)
176:  {
177:    int ret;
178:                                  /* 自分のウィンドウかどうかを調べる */
179:    if (windowptr == (window *)(eventrec.whom))
180:      {
181:                                  /* <graph> をカレントにセット */
182:        GMSetGraph ((graph *)(windowptr));
183:        GMMove (16 << 16 | 16);
184:        GMDrawStrZ ("Input");
185:        TMCaret (teHdl, 0);       /* カーソルを消す */
186:        WMUpdate (windowptr);     /* アップデートリージョンをセット */
187:        TMUpDate (teHdl, &terect);/* <tEdit> の文字列を表示 */
188:        GMMove (16 << 16 | 48);
189:        GMDrawStrZ ("Output");
190:        display ();
191:        if (initflag == 0)
192:          {
193:            TMSetSelect (teHdl, (**teHdl).length, (**teHdl).length, (**teHdl).length);
194:            initflag = 1;
195:          }
196:        CMDraw (windowptr);       /* ボタンを表示 */
197:        WMUpdtOver (windowptr);   /* リージョンを戻す */
198:      }
199:  }
200:
201:  /*
202:  ** アクティベートイベント
203:  */
204:
205:  void activateEvent (void)
206:  {                               /* 自分のウィンドウかどうかを調べる      */
207:    if (windowptr == (window *)(eventrec.whom))
208:      {
209:        activflag = 1;
210:        eventmask = 0x32ab;       /* イベントマスクをセット */
211:      }
212:    else
213:      {
214:        if (activflag != 0)
215:          {
216:            GMSetGraph ((graph *)(windowptr));
217:            TMCaret (teHdl, 1);
218:            activflag = 0;
219:            eventmask = 0x3282;   /* イベントマスクをセット */
220:          }
221:      }
222:  }
223:
224:  /*
225:  ** タスクマンのイベント
226:  */
227:
228:  void app12Event (void)
229:  {
230:    int sendtask;
231:    int ret, i;
232:
233:    switch (eventrec.what2)
234:      {
```

```
235:         case CLOSEALL:             /* 全ウィンドウのクローズ */
236:         case ENDTSK:               /* 全タスクの終了 */
237:           EndPr (0, 0);            /* 終了する */
238:           break;
239:         case WINDOWSELECT:         /* ウィンドウのセレクト */
240:           WMSelect (windowptr);    /* ウィンドウをアクティブにする */
241:           break;
242:       }
243:   }
244:
245:   /*
246:   ** 初期処理
247:   */
248:
249:   int Init (void)
250:   {
251:     task tbuff;
252:     int ret;
253:
254:     TSSetAbort (&EndPr, 0);        /* アボート処理ルーチン登録 */
255:     taskid  = TSGetID ();          /* タスク ID を得る */
256:     TSGetTdb (&tbuff, -1);         /* タスク管理テーブルを得る */
257:     if ((sxver = (short)SXVer ()) < 0x102)
258:       {
259:         DMError (0x0001, "ＳＸシステムのバージョンが違います。");
260:         return (-1);
261:       }
262:     ret = TSTakeParam (&tbuff.command, &wsize, 0, 0, 0);
263:     if ((ret & 1) == 0)
264:       {
265:         *(long *)(&wsize.left) = TSGetWindowPos ();
266:         wsize.right = wsize.left + 256;
267:         wsize.bottom = wsize.top + 80;
268:       }
269:     ret = WMOpen ((window *)(0), &wsize, (LASCII *)(&wtitle), 0, 32<<4,
270:                   (window *)(-1), -1, taskid);
271:     windowptr = _SXCALLPtr;
272:     if (ret < 0) return (ret);
273:     GMSetGraph ((graph *)(windowptr));
274:     GMFontKind (1);                /* 16 ドットフォント */
275:     GMFontMode (0);
276:     ret = TMNew2 (&terect, &terect, (graph *)(windowptr));
277:     teHdl = _SXCALLPtr;
278:     if (ret < 0) return (ret);     /* <tEdit> レコードが作成できなかった */
279:     (**teHdl).lenMax = BUFFMAX;
280:     (**teHdl).lineHeight = 16;     /* 改行幅をセット */
281:     WMShow (windowptr);            /* ウィンドウを表示する */
282:     return (0);                    /* 初期化成功 */
283:   }
284:
285:   /*
286:   ** 終了処理
287:   */
288:
289:   void EndPr (int code, long param)
290:   {
291:     if (windowptr != 0)
292:       {
293:         CMKill (windowptr);        /* コントロールを廃棄 */
294:         WMDispose (windowptr);     /* ウィンドウを廃棄 */
295:         TMDispose (teHdl);         /* <tEdit> レコードを廃棄 */
296:       }
297:     exit (code);                   /* タスクを終了する */
```

```
298:  }
299:
300:  void display (void)
301:  {
302:    rect frect = {80, 48, 80 + 8 * BUFFMAX, 48 + 16};
303:    int ret;
304:
305:    ret = GMPenMode (1 << 8);
306:    GMFillRect (&frect);
307:    GMPenMode (ret);
308:    GMMove (80 << 16 | 48);
309:    GMDrawStrZ (outbuf);
310:  }
311:
312:  void readch (void)
313:  {
314:    do {
315:      if ((ch = *inbufptr++) == NULL) return;
316:    } while (ch == ' ' || ch == '\t');
317:  }
318:
319:  void expression (void);
320:
321:  void factor (void)
322:  {
323:    if (ch == '(') {
324:      readch ();
325:      expression ();
326:      if (ch == ')')
327:        readch ();
328:      else
329:        *outbufptr++ = '?';
330:    } else if (isgraph (ch)) {
331:      *outbufptr++ = (char)ch;
332:      readch ();
333:    } else
334:      *outbufptr++ = '?';
335:  }
336:
337:  void term (void)
338:  {
339:    factor ();
340:    for (;;)
341:      if (ch == '*') {
342:        readch ();
343:        factor ();
344:        *outbufptr++ = '*';
345:      } else if (ch == '/') {
346:        readch ();
347:        factor ();
348:        *outbufptr++ = '/';
349:      } else
350:        break;
351:  }
352:
353:  void expression (void)
354:  {
355:    term ();
356:    for (;;)
357:      if (ch == '+') {
358:        readch ();
359:        term ();
360:        *outbufptr++ = '+';
361:      } else if (ch == '-') {
```

```
362:         readch ();
363:         term ();
364:         *outbufptr++ = '-';
365:       } else
366:         break;
367:   }
368:
369:   void polish ()
370:   {
371:     inbufptr = inbuf;
372:     outbufptr = outbuf;
373:     readch ();
374:     expression ();
375:     while (ch) {
376:       *outbufptr++ = '?';
377:       readch ();
378:     }
379:     *outbufptr = '\0';
380:     return;
381:   }
```

◆ プログラムのコンパイル

サンプルプログラムをコンパイルするには，コンパイラにデバッグ情報を生成するオプション“-g”および SX–Window プログラムであることを **GCC** に指定する“-SX”オプションを指定します。このとき“-O”以外の最適化オプションは，指定しないようにしてください。それでは，次のように入力してコンパイルしてみましょう。

```
A>gcc -g -SX -O test.c
```

5.2.2 GDB の起動

GDB はコマンド対話型のデバッガですから，**GDB** の出力は標準出力に対して行われます。このため，SX–Window 上で標準出力を実行すると，画面が乱れてしまう場合もあります。

Human68k 版 **GDB** には画面を乱すことなくデバッグできるように，ターミナルを使用したリモートコンソールモード[6]と，**GDB** の画面とアプリケーションの画面を切り替えるスクリーンスワップモード[7]の 2 通りの方法が用意されています。このセクションでは，とりあえずスクリーンスワップモードで説明していきます。詳しくは，第 5.3 節「**GDB** の起動 (P.184)」を参照してください。

それでは，次のように入力して **GDB** を起動してください[8]。

```
A>gdb -swap test.x
```

GDB が起動すると，次に示す画面が現れます。最後の行の“(gdb)”が **GDB** のプロンプトです。ユーザはこのプロンプトに対してコマンドを入力することで，**GDB** に指示することができます。また，このプロンプトが表示されている状態が，コマンドを受けつける状態になっていることを示しています。

6)'-remote' を参照 (第 5.3.5 節 P.185)。

7)'-swap' を参照 (第 5.3.5 節 P.186)。

8)この時点で FSX.X がメモリに常駐していなければなりません。

```
GDB is free software and you are welcome to distribute copies
 of it under certain conditions; type "show copying" to see
 the conditions.
There is absolutely no warranty for GDB; type "show warranty"
 for details.
GDB 4.* X*_**, Copyright 1991 Free Software Foundation, Inc.
(gdb)
```

5.2.3 GDB を使う

それでは実際に，**GDB** を操作して実行ファイル test.x をデバッグしてみましょう。

◆ ソースプログラムを表示する

“list”，または“l”と入力してください[9]。ソースプログラム中の次に実行される行を中心に 10 行だけ表示されます[10]。画面の左端には行番号が表示され，次にその行に対応するソースコードが表示されます。ユーザはこの行番号を **GDB** に指定することで，ブレークポイントなどの設定を行うことができます。

9)‘l’は‘list’の省略形です。以降コマンドに省略形があれば，そのつど説明します。

10)第 5.9.1 節 (P.217) で詳しく説明しています。

```
(gdb) list [return]
38 int ch;
39 char wtitle[] = "\017Polish notation";  /* ウィンドウタイトル*/
40 rect terect = {16+68+16, 16, 260, 32};  /* <tEdit> の dest */
41 common char _sxkernelcomm[] = "sxwdb.x -D -K -L0";
42                                          /* カーネル起動コマンド */
43 int main ()
44 {
45 int     ret;
46
47 if ((ret = Init()) < 0)
(gdb)
```

今度は ⏎ だけを押してください。すると，前に表示したソースプログラムの次に続く 10 行が表示されます。さらに，⏎ だけを押してください。すると，また次に続く 10 行が表示されます。

```
(gdb) [return]
48   EndPr (ret, 0);            /* 初期化に失敗したので終了する */
49 while (1) {                  /* タスクマンのイベントを得る   */
50     TSEventAvail (eventmask, &eventrec);
51                              /* 同時にタスクの切り替えが行われる */
52     switch (eventrec.what) {
53       case E_IDLE:
54         nullEvent ();
55         break;
56       case E_MSLDOWN:
57         mouseLDownEvent ();
(gdb) [return]
58         break;
59       case E_MSRDOWN:
60         mouseRDownEvent ();
61         break;
62       case E_KEYDOWN:
63         keyDownEvent ();
64         break;
65       case E_UPDATE:
66         updateEvent ();
67         break;
(gdb)
```

このように ↵ だけを押していくと，前に入力したコマンドを繰り返し実行することができます。この繰り返し実行機能は他のコマンドに対しても有効ですが，いくつかのコマンドでは無意味な場合もあります。つまり，すべてのコマンドに対して有効なわけではありません。

また,「↵ を押すごとに10行ずつ表示される」と説明しましたが，これはデフォルト状態のことであり，表示行はコマンドの設定しだいで何行にでも変更することができます[11]。

11)第5.13.11節(P.232)を参照。

次に， list コマンドの他の使い方について説明します。まず，

```
list Init
```

と入力して ↵ を押してください。この場合は，Init () 関数の始まりを中心とした10行が表示されます。関数名の代わりに行番号を入力すれば，指定した行番号を中心とした10行を表示させることもできます。

```
(gdb) list Init [return]
245 /*
246 **  初期処理
247 */
248
249 int Init (void)
250 {
251   task    tbuff;
252   int     ret;
253
254   TSSetAbort (&EndPr, 0);     /* アボート処理ルーチン登録 */
(gdb)
```

また，複数のソースファイルをリンクしたプログラムの場合，

```
list <ファイル名> : <関数名>
```

または，

```
list <ファイル名> : <行番号>
```

と入力すると，表示したいファイルを指定することもできます。

◆ ブレークポイントを設定する

通常は，バグがあると思われる位置にブレークポイントを設定しますが，今回は，とりあえず main () 関数に設定します[12]。次のように入力してみてください。

```
break main
```

すると，次のメッセージが表示されます[13]。

```
(gdb) break main [return]
Breakpoint 1 at 0x5d0: file test.c, line 47.
(gdb)
```

このメッセージは，1 番のブレークポイントをメモリ上の 0x5d0 番地，ソースファイル test.c の 47 行目に設定していることを表しています。ここで，メモリ上の 0x5d0 番地というのはおかしいと思われるかもしれません。しかしこれは，デバッグ対象プログラムがまだロードされていない状態なので，ここではプログラム先頭からのオフセットが表示されていると考えてください[14]。

12)第 5.7 節 (P.207) 参照。

13)'break' コマンドの省略形は 'b' です。

14)プログラムがロードされた後は，メモリの絶対アドレスを表示するようになります。

15)'r' は 'run' の省略形です。

◆ デバッグ対象プログラムの実行

"run" または "r"[15]と入力してみてください。GDB はこのコマンドでデバッグするプログラムをロードし，実行します。プログラムの実行が進み，設定しておいたブレークポイントに達すると，次のメッセージが表示され，プログラムの実行が停止します。

```
Breakpoint 1, main () at test.c:47
47   if ((ret = Init ()) < 0)
(gdb)
```

これで，main () 関数の入り口で停止したことになります。また同時に，次に実行されるソースコードが表示されます。

16)'s' は 'step' の省略形です。

◆ ステップ実行

"step" または "s"[16]と入力してください。ソースコードが 1 行だけ実行されて，停止します。ソースコードの 47 行目は Init () 関数を呼び出しますので，その結果，Init () 関数の始めで停止することになります。

```
(gdb) step [return]
Init () at test.c:254
254    TSSetAbort (&EndPr, 0);          /* アボート処理ルーチン登録 */
(gdb)
```

ここで，⏎ を 2，3 回押してみてください。ソースコードが 1 行ずつ実行されていきます。

```
(gdb) [return]
255    taskid  = TSGetID ();             /* タスク ID を得る */
(gdb) [return]
256    TSGetTdb (&tbuff, -1);            /* タスク管理テーブルを得る */
(gdb) [return]
257    if ((sxver = (short)SXVer ()) < 0x102) {
(gdb)
```

17)第 5.4.4 節 (P.190) を参照。

再度 "step 10 (または s 10)"[17]と入力して，⏎ を押してください。今度は，ソースコードを 10 ステップ実行して停止します。また "step" コマンドでは，引数にリピート回数を指定することができます。

さらに "step" と同じようなコマンドに，"next" コマンドがあります。このコマンドは，ソースコードが 1 行ずつ実行される点では "step" コマンドと同じですが，関数呼び出しがある場合は関数内を一度に実行します。つまり，関数内をステップ実行する必要がないときに便利なコマンドです。

◆ データを調べる

参照可能なグローバル変数名とスタティック変数名を表示してみましょう。まず，次のように入力してみてください。

```
info variables
```

すると，次のように表示されるはずです[18]。

18)「グローバル変数を調べる」を参照 (第 5.5 節 P.193)。

```
(gdb) info variables [return]
All defined variables:

File _globals_:
char _sxkernelcomm[18];
int activflag;
int ch;
int eventmask;
struct tsevent eventrec;
char inbuf[21];
char *inbufptr;
int initflag;
char outbuf[21];
char *outbufptr;
int sxver;
int taskid;
struct tEdit **teHdl;
struct rect terect;
struct window *windowptr;
struct rect wsize;
char wtitle[17];

Non-debugging symbols:
        00001774  __size_info
        00001804  _doserrno
        00001808  errno
(gdb)
```

それでは，変数 sxver を調べてみましょう。変数の内容を表示するには，“print” コマンドを使います。このコマンドは，プログラムを書くのと同じ形式で式を入力することで，式の値を調べることができます。

```
print sxver
```

または，

```
p sxver
```

と入力してください[19]。SX–Window Ver. 2.1 であれば，次のように表示されます。

19)「変数の内容を表示する」を参照 (第 5.6 節 P.197)。

```
(gdb) print sxver [return]
$1 = 513
(gdb)
```

ここでは，変数 sxver の値は "513" ということになります。"$1 = " というのは，「GDB 内部の変数履歴番号 1 に変数の値 "513" が記録された」ということを意味します。"print" コマンドで変数を表示していくと，その値は GDB 内部の変数履歴に次々と記録されていきます。変数履歴に記録された値は，その変数履歴番号を指定することでいつでも参照することができます。また，変数は値が変化していくものです。そのため，値が変化する前に print コマンドで表示しておけば，値が変化しても変数履歴から前の値を参照することができます[20]。

20)第 5.6.4 節 (P.204) を参照。

◆ 表示フォーマットの変更

変数 sxver に代入された値は SX-SYSTEM のバージョンナンバーを得るシステムコールによって得た値ですが，この値は上位バイトにメジャーバージョン，下位バイトにマイナーバージョンが入っています。つまり，値が 16 進数で表示されているほうがわかりやすいということです。"print" コマンドで整数型の変数を参照した場合，通常は値を 10 進数で表示します。これを 16 進数で表示するには，

```
print/x sxver
```

と入力します。すると，次のように 16 進数で表示されます[21]。

21)第 5.6 節 (P.197) を参照。

```
(gdb) print/x sxver [return]
$2 = 0x201
(gdb)
```

◆ 変数の値を変更する

変数の値を変更するには "set" コマンドを使います。たとえば変数 sxver の値を "0x110" に変更するには，

```
set sxver = 0x110
```

と入力します。また "print" コマンドを使って，

```
print sxver = 0x110
```

と入力しても，同様に値を変更することができます。ただし，print コマンドの場合，値が変更された後に変数の値を表示します。

この例では 16 進数表記を使用しましたが，10 進数表記でもまったく同様です。

```
(gdb) print sxver [return]
$1 = 0x110
(gdb)
```

◆ 構造体のデータを調べる

次に，構造体型の変数 tbuff を表示してみましょう。変数 sxver のときと同様に，“print” コマンドを使用します。それでは，

```
print tbuff
```

と入力してみてください。画面は次のようになります。

```
(gdb) print tbuff [return]
$2 = {name = {"B:\\GDB\\TEST.X", '\000' <repeats 77 times>},
  command = {length = 0 '\000', Lstr = {
      '\000' <repeats 255 times>}}, tskid = 1, parentid = 0,
  stmode = 4, rsctype = 1615200, rscid = 0, state = 1,
  programPtr = 0x18a560, programHdl = 0x0, dataHdl = 0x56e6d0,
  envPtr = 0xa19de "", regkeep = {0, 0, 0, 0, 0, 0, 0,
    1710882, 1807752, 661982, 0, 0, 0, 0, 1, 0, 1710882,
    1615342}, commid = -1, rsv = {-1, 0, 26139, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, -1, -1, 0 <repeats 16 times>}}
(gdb)
```

このように構造体の要素数が多く，複雑なために大変見にくい画面になっています。次に，この画面を見やすくする方法を説明しましょう。まず，

```
set print pretty on
```

と入力してください。次に，

```
print
```

と入力します。

2 回目に，ただ単に “print” と入力しているのは，直前に表示した式を再評価して，新たに表示することを意味しています。このように “set print pretty on” と入力すると，構造体を 1 つのメンバが 1 行にインデントされたフォーマットで表示することができます[22)]。また構造体を最初のようにコンパクトなフォーマットで表示するには，“set print pretty off” と入力します。

22)第 5.6 節 (P.197) を参照。

```
(gdb) set print pretty on [return]
(gdb) print [return]
$3 = {
  name = {"B:\\GDB\\TEST.X", '\000' <repeats 77 times>},
  command = {
    length = 0 '\000',
    Lstr = {'\000' <repeats 255 times>}
  },
  tskid = 1,
  parentid = 0,
  stmode = 4,
  rsctype = 1615200,
  rscid = 0,
  state = 1,
  programPtr = 0x18a560,
  programHdl = 0x0,
  dataHdl = 0x56e6d0,
  envPtr = 0xa19de "",
  regkeep = {0, 0, 0, 0, 0, 0, 0, 1710882, 1807752, 661982, 0,
    0, 0, 0, 1, 0, 1710882, 1615342},
  commid = -1,
  rsv = {-1, 0, 26139, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, -1,
    0 <repeats 16 times>}
}
(gdb)
```

◆ ポインタからデータを参照する

SX–Window のシステムコールには，WMOpen () 関数のように構造体へのポインタを返すものがあります。この場合，構造体はシステムの中で作られるため，ユーザプログラムからその構造体を参照するには，構造体へのポインタを使ってアクセスすることになります。サンプルプログラムでは，WMOpen () 関数の返すwindow 構造体へのポインタを変数 windowptr に格納するようになっています。それでは，この変数 windowptr を調べてみましょう。ただし，その前にプログラムの実行が 271 行まで進んでいることを確認してください。まず，

```
print windowptr
```

と入力してみてください。画面は次のようになります。

```
(gdb) print windowptr [return]
$4 = (struct window *) 0x54abd6
(gdb)
```

変数 windowptr は window 構造体へのポインタですから，変数 windowptr の内容にアドレスが入っているのは当然の結果です。また，このポインタの指す構造体の中身を表示するには，変数名の前に "*" をつけます。次のように，

```
print *windowptr
```

と入力してみてください。画面は次のようになります。

```
(gdb) print *windowptr [return]
$5 = {
  wGraph = {
    bmap = 0x54abb0,
    grRect = {
      left = 0,
      top = 0,
      right = 256,
      bottom = 80
    },
    procs = 0x0,
    visible = 0x56e748,
    clipping = 0x56e6e8,
    drawLvl = 0,
    penMode = 0,
    penLoc = 0,
    penSize = 65537,
    penPat = 0xe031e,
    exPat = 0xe063e,
    workKind = 0,
    workHdl = 0x0,
    fgColor = 65535,
    bgColor = 9,
    fontKind = 1,
    fontFace = 0,
    fontMode = 0,
    fontSize = 0
        :
(gdb)
```

23)第 5.4 節 (P.189) を参照。

24)main () 関数の 49 行目です。

25)「実行の再開」を参照 (第 5.4.3 節 P.190)。

◆ 関数からリターンする

もし，デバッグ中の関数をこれ以上デバッグする必要がなければ，“finish” コマンドを使います。このコマンドは関数の途中から終わりまでを一度に実行し，自動的に現在の関数を呼び出した位置へ戻ります[23)]。それでは，

```
finish
```

と入力してみましょう。すると，現在実行中の Init () 関数を終わりまで一度に実行し，Init () 関数から戻った地点[24)]で停止します。

次は，polish () 関数をデバッグしてみます。まず，polish () 関数にブレークポイントを設定します。次のように入力して，実行してみましょう。

```
break polish
```

このように，次々とブレークポイントを設定してプログラムをデバッグしていきます。

◆ 実行の再開

一通り，デバッグ作業が完了したらプログラムの実行を再開します。プログラムの実行を再開するコマンドは，“continue” です。次のように入力してください。

```
continue
```

これでデバッガからデバッグ対象プログラムに制御が移り，プログラムの実行が再開されます[25)]。しかしこの時点では，サンプルプログラムが動作しており，イベントを待っている状態です。ここで，サンプルプログラムに対して，

```
(A+B)*(C-D)
```

と入力して，↵ を押してみてください。すると，デバッガに制御が移り，371 行で停止します。

◆ 配列のデータを調べる

入力したデータを調べてみましょう。さきほど入力した “(A+B)*(C-D)” というデータは，SX–SYSTEM のテキストマンによって char 型の配列 inbuf に格納されているはずです。

```
print inbuf
```

と入力してみてください。次の画面のように，char 型の配列のデータは文字列として表示されます。

```
(gdb) print inbuf [return]
$6 = {"(A+B)*(C-D)\000\000\000\000\000\000\000\000\000\000"}
(gdb)
```

◆ デバッグを終了する

SX-Window アプリケーションのデバッグをしているときは，デバッグ対象プログラムと SX シェルを終了させてから **GDB** を終了させてください[26]。次に，そのための手順を示します。

26)現在の GDB では，SX-Window アプリケーションを強制終了させることができません。

1. “`delete`” コマンドですべてのブレークポイントを削除する
2. “`cont`” コマンドでプログラムの実行を再開する
3. ウィンドウのクローズボタンをクリックしてプログラムを終了する
4. システムアイコンから終了を選択し SX シェルを終了する
5. “`quit`” コマンドで **GDB** を終了する

5.3 GDB の起動と終了

ここでは，デバッグ対象プログラムを **GDB** でデバッグする前に，あらかじめ準備しておかなければならない点について説明し，次に，**Human68k** のコマンドラインから **GDB** を起動する方法を説明します。そして最後に，**GDB** を終了する方法について説明します。

5.3.1 プログラムをデバッグできるように準備する

GDB でプログラムをデバッグするには，プログラムをコンパイル／リンクする際に，必要なすべてのデバッグ情報を生成するようにコンパイラに指示する必要があります。

GCC を使った場合は，コンパイラに対して“-g”オプションを指定してコンパイルします。このとき，同時に最適化オプションを指定することもできますが，“-fomit-frame-pointer”オプションは指定しないほうがいいでしょう。このオプションを指定すると，コンパイラは必要のないスタックフレームを生成しなくなりますので，ローカル変数の参照ができなくなる恐れがあります。また，デバッグしたい関数はインライン展開しないようにしてください。なぜならば，インライン展開した関数はステップ実行できなくなるからです。

5.3.2 GDB が使うファイル

プログラムのデバッグには，次に示すファイルが必要です。

❖ **ソースファイル**

デバッグ対象プログラム[1]の C で書かれたファイルのことです。

1)以下，プログラムと表記する。

❖ **実行ファイル**

プログラムのソースファイルをコンパイルして作成されたファイルのことです。このファイルには，シンボリックデバッグ情報が含まれていなければなりません。

5.3.3 GDB が使う環境変数

GDB は以下に示す環境変数を使用します。ただし，必ずしもこれらを必要とはしません。

❖ "HOME"

GDB が起動するときに読み込む "`.gdbinit`" ファイル[2] がおかれているディレクトリをフルパスで指定します。

❖ "GDB_OPTION"

GDB を起動するときに指定するコマンドラインオプションを，そのままこの環境変数で指定することができます。

2)Ext 氏の Twenty-One が常駐していなければ '_gdbinit' です。第 5.12.3 節 (P.226) を参照。

また，チャイルドプロセスの起動方法の変更により，設定する環境変数が追加されました[3]。**X680x0 GDB** がチャイルドプロセスの起動に使用するコマンドシェルは，環境変数 "SHELL"，"SHELL_OPT"，"SHELL_TYPE" に設定します。設定しなかった場合は `command.x` が使用されます。この場合，`command.x` のディレクトリは環境変数 PATH に設定されていなければなりません。

3)詳しくは「X680x0 Develop. & libc. II」を参照。

5.3.4 起動方法と書式

GDB はコマンドラインから次のような書式で起動します。

```
gdb [<オプションスイッチ>] <ファイル名>
```

<オプションスイッチ> には，引数を指定します (省略可能)。そして，<ファイル名> には，プログラムの名前を "`.x`" 拡張子を含めて指定します。また **SHARP** のデバッガ "`db.x`" などのように，プログラムに渡すコマンドラインを続けて指定することはできません[4]。

4)理由については，第 5.4 節 (P.189) の「プログラムの実行」で説明します。

5.3.5 オプションスイッチ

GDB では，起動時に次のオプションスイッチが使用できます。

- "`-b`" リモートデバッグ時の通信速度の指定[5]
- "`-batch`" バッチ処理
- "`-cd`" ワーキングディレクトリの指定
- "`-command`" コマンドファイルの指定
- "`-core`" コアダンプファイルの指定[6]
- "`-directory`" ソースディレクトリの指定
- "`-exec`" デバッグ対象プログラムの指定[7]

5)Human68k版 GDB では意味がありませんし，使用することもできません。なお X680x0 GDB では，廃止されています。

6)Human68k版 GDB では意味がありませんし，使用することもできません。なお X680x0 GDB では，廃止されています。

7)実行ファイルとシンボル情報ファイルが分かれているときに使います。なお X680x0 GDB では，廃止されています。

- “-epoch”，“-fullname”　Emacsを使用する[8]
- “-help”　ヘルプメッセージの表示
- “-mem”　メモリの設定
- “-nx”　初期化ファイル“.gdbinit”[9]を読み込まない
- “-quiet”　タイトルの非表示
- “-remote”，“-r”　リモートコンソールモードで起動する
- “-se”　デバッグ対象プログラム名の指定[10]
- “-symbols”　シンボル情報のファイルの指定[11]
- “-swap”　スクリーンスワップモードで起動する
- “-tty”　標準入出力先の指定

8)Human68k版GDBでは意味がありません。なおX680x0 GDBでは，廃止されています。

9)第5.12.3節(P.226)を参照。

10)X680x0 GDBでは，廃止されています。

11)実行ファイルとシンボル情報ファイルが分かれているときに使います。なおX680x0 GDBでは，廃止されています。

5.3.6　GDBのコマンド入力

ここでは，**GDB**が起動した後のコマンドの入力について説明します。

◆ コマンド入力

GDBはコマンドの受け入れ準備ができると，プロンプト“(gdb)”[12]を表示します。ユーザはこのプロンプトに続いて，コマンド名とそのコマンドの引数を入力することで，**GDB**に命令することができます。**GDB**はコマンドの実行が終了すると，再びプロンプトを表示し，ユーザからのコマンド入力を待ちます。

12)これはデフォルトの設定です。プロンプトはユーザが自由に変更することができます(第5.13.8節(P.230)の‘set prompt’コマンドを参照)。

◆ コマンド行編集

コマンド入力中は，テキストエディタ感覚で行編集が行えます。つまり，入力中の文字列をカーソルキーを使ってカーソルを移動し，文字を挿入したり削除するといったことができます。またヒストリ機能[13]を使えば，前に実行したコマンドを呼び出して編集／実行することもできます。

13)前に入力したコマンドを使用することができます(Vol. 2, P.211参照)。

◆ コマンド名の省略形

GDBのコマンドはほとんどが長い名前なので，コマンド名をすべて入力しなくても，1文字や少ない文字数の省略形で入力できるようになっています。コマンドはあいまいにならないかぎりで，省略することができます。もしも入力したコマンドの省略形が，複数のコマンドにマッチする場合は，それらのコマンド名をすべて表示します。ただし，あいまいな省略形を特別に許可しているものも存在します。たとえば，省略形の“**s**”は，他の“**s**”で始まるコマンドをさしおいて“**step**”コマンドを意味するように設定してあります。

◆ コンプリーション機能

GDBでは，コンプリーション機能が使用できます。コンプリーション機能とは文字列を完全に入力しなくても，入力途中で“CTRL + I”[14]を押せば，**GDB**のほうで現在参照可能なシンボル名から検索し，ユーザが途中まで入力した文字列を完全な文字列に完成してくれる機能のことで，文字数の多いシンボル名[15]を

14)コントロールキーを押しながら‘I’を押します。

15)関数，変数名のことです。

入力する際にとても便利です。この機能は **GDB** のコマンド名に対しても有効ですが，コマンド名の入力には，やはり省略形を使ったほうが便利でしょう。

◆ 繰り返し実行

GDB に空行[16]を入力するということは，直前のコマンドをそのまま繰り返すことを意味します。ただし一部のコマンドについては，この方法による繰り返しを許可していません。なぜならば，無意識による繰り返しがトラブルを引き起こしたり，繰り返し実行することが無意味なコマンドなどがあるからです。また，別のあるコマンドでは，繰り返されたときにさらに便利なように，異なった動作をします。たとえば，ステップ実行のコマンド“`step`”は，一度実行した後は，↵ を押していくだけでステップ実行を進めていくことができるようになっています。

16)プロンプトに対して何も入力しないで，そのままリターンキーを押します。

5.3.7 プログラムの実行

ここでは，プログラムを **GDB** に対して指定する方法と，そのプログラムのデバッグを開始するためにプログラムを実行させる方法について説明します。

◆ プログラムの指定

デバッグを始めるには，まず **GDB** にシンボリックデバッグ情報を読み込ませる必要があります。そのためには，デバッグするプログラムのファイルを指定しなければなりません。プログラムを指定する方法には，次の 2 種類があります。

❖ **起動時に引数によるファイル名の指定**

GDB のコマンドラインオプションにファイル名を指定する場合は，次のように入力して **GDB** を起動します。

```
gdb <ファイル名>
```

この場合は，起動時にプログラムのシンボル情報が読み込まれます。

❖ **起動後のコマンドによるファイル名の指定**

通常は，**GDB** 起動時にプログラムを指定しますが，プログラムが複数個あった場合など，別のプログラムへと変更することがあります。また，**GDB** 起動時にデバッグしたいプログラムを記述し忘れたり，ファイル名をまちがえて記述したりすることもあります。これらの場合，いちいち **GDB** を終了させて起動し直すのはめんどうですから，新しくプログラムを指定します。コマンドによるファイル指定には，次に示す 3 通りの方法があります。

- “`exec-file <ファイル名>`”
  プログラム名を指定します。もしも，ファイルをワーキングディレクトリで見つけることができなかった場合，**GDB** は環境変数 `PATH` に従ってコマンド検索対象ディレクトリからファイルを探します。つまり，カレントディレクトリにプログラムが存在しなくてもよいということです。また，ファイル名を絶対／相対パスで指定することもできます。

- “symbol-file <ファイル名>”
  ファイルからシンボルテーブルの情報を読み込みます。このコマンドも“exec-file”コマンドと同じく，ワーキングディレクトリで見つからなければ環境変数 PATH に従って検索します。
- “file <ファイル名>”
  このコマンドは，“exec-file”コマンドと“symbol-file”コマンドを一度に実行させるためのコマンドです。通常，プログラムを指定する場合は，このコマンドを使います。

5.3.8 GDB を終了する

GDB を終了するコマンドは，“quit”です。プロンプトに対して

```
quit
```

と入力することで，**GDB** を終了させることができますが，SX–Window アプリケーションのデバッグ中はこのコマンドを使ってはいけません。必ず，SX–Window アプリケーションを終了し，SX シェルも完全に終了させてから“quit”コマンドで **GDB** を終了させるようにしてください[17]。

17)GDB はプロセスを強制終了させるために DOS コールの EXIT を使用しているので，SX–Window アプリケーションを終了させることができません。

5.4 ····················· プログラムの実行を制御する

GDB には，プログラムの実行を制御するために，次のような方法が用意されています。

- **プログラムの実行**
- **停止したプログラムの実行の再開**
- **ソースファイルを行単位で，またはマシン命令を命令単位で実行する**
- **関数の呼び出しをスキップする**
- **現在の関数から呼び出した関数に戻るまで実行する**
- **プロセスの削除**
- **実行中のプログラムの中断**

このセクションでは，プログラムの実行を制御するためのさまざまなコマンドについて説明します。

5.4.1 プログラムの実行

最初にプログラムを実行させるには “run”[1] コマンドを使います。このコマンドは，デバッグするプログラムのロードと実行を行うためのもので，引数には，デバッグするプログラムに与えるコマンドライン引数が指定できます[2]。第 5.3 節 (P.184)「**GDB の起動**」のところで，**GDB** 起動時のコマンドライン引数に，プログラムのコマンドライン引数を指定できないと説明したのはこのためです。

1)省略形は ‘r’ です。

2)引数は必要なければ省略できます。

―― ■ 書式 ■ ――

■ `run [<引数> ... ]`
プログラムを実行させる

5.4.2 プログラムの再スタート

デバッグを最初からやり直したいときは，“run”コマンドで再実行させることができます[3]。この場合デバッガは，現在デバッグしているプログラムを削除し，新たにプログラムを読み込み，実行します。ただし SX-Window アプリケーションの場合は，デバッグしているプログラムを終了させてから行ってください。

3)プログラムに渡すコマンドラインオプションが前に設定したものと同じならば，再び指定する必要はありません。

5.4.3 実行の再開

ブレークポイントで停止したプログラムの実行を再開するには，“continue”[4]コマンドを使います。このコマンドはプログラムの実行が進み，ブレークポイントに出会うかプログラムが終了するまで，実行を停止しません。

4)省略形は‘c’です。

―― ■書式■ ――

- continue [<カウント>]
  ブレークポイントで停止したプログラムの実行を再開する

また，continue コマンドと同様な機能をもつコマンドに“jump”[5]があります。このコマンドの場合，引数に実行を再開させたい「行番号」または「アドレス」を指定することができます[6]。つまり，実行させたくない部分を飛ばすことが可能なのです。ただし，現在実行中の関数またはブロックを超えて指定した場合は，思いがけない結果になってしまいます。そのため，指定された行が現在実行中の関数の外であれば，jump コマンドは確認を求めてきます。

5)省略形は‘j’です。

6)このコマンドは引数を省略することができません。

―― ■書式■ ――

- jump <行番号>
  <行番号> から実行を再開する
- jump *<アドレス>
  <アドレス> から実行を再開する[7]

---

7)GDB のコマンドに対して引数にアドレスを指定する場合は‘$*’をつけてください。

5.4.4 ステップ実行

ソースコードを 1 行単位で実行するには“step”[8]コマンドを使います。このコマンドに何も引数をつけなければ，ソースコードを 1 行実行したところで停止しますが，引数にリピート回数を指定することで，一度に複数行実行することができます。また，“stepi”[9]コマンドでマシン命令を 1 命令単位で実行させることもできます。

8)省略形は‘s’です。

9)省略形は‘si’です。

■ 書式 ■

- **step [<リピート回数>]**
  ステップ実行する
- **stepi [<リピート回数>]**
  マシン命令を 1 命令単位でステップ実行する

関数の呼び出しをスキップしながらソースコードを 1 行単位で実行するには，“**next**”[10]コマンドを使います。このコマンドは，次に実行するソースコードに関数呼び出しがあり，かつその関数をステップ実行させる必要がないときに便利です。また，“**nexti**”[11]コマンドでマシン命令を 1 命令単位で実行させることもできます。

10)省略形は ‘n’ です。

11)省略形は ‘ni’ です。

■ 書式 ■

- **next [<リピート回数>]**
  関数呼び出しをスキップしながらステップ実行する
- **nexti [<リピート回数>]**
  マシン命令を 1 命令単位でステップ実行する

さらに，次に示すコマンドは，ステップ実行をサポートする非常に便利なコマンドです。

■ 書式 ■

- **finish**[12]
  現在の関数から，呼び出し側のルーチンに戻るまでプログラムを実行する
  このコマンドは，ステップ実行をしているときにまちがえて関数の中に入り込んだ際，その関数から抜け出す場合に便利である
- **until**[13]
  ステップ実行でループに入ってしまったときなどに，プログラムカウンタがジャンプアドレスより大きくなるまで自動的にプログラムの実行を継続する。つまり，ループから抜け出すことができる

---

12)省略形は ‘fin’ です。

13)省略形は ‘u’ です。

5.4.5 プロセスの削除

14)省略形は 'k' です。

"kill"[14] コマンドは，デバッグ中のプログラムのプロセスを強制削除します。ただし，SX-Window アプリケーションの場合は使用してはいけません。

■書式■

■ kill

プロセスを削除する

5.4.6 実行中のプログラムの中断

無限ループに入ってしまったとか，予想できない理由でデバッガに制御が戻らないときのために，**X68000** 本体の INTERRUPT スイッチを押すことで強制的にプログラムの実行を停止させることができます。ただし，必ずデバッガに制御が戻るという保証はありません。

5.5 ································ デバッグ状態の調査

ここでは，デバッガの制御のもとでプログラムの状態を調べる方法について説明します。

デバッガは，プログラムの状態を調べるために次のようなことができます。

- **プログラムのソースファイルを調べる**
- **プログラムのコマンドライン引数を調べる**
- **関数を調べる**
- **変数を調べる**
- **ソースコードに対応するメモリ上のアドレスを調べる**

5.5.1 デバッグ状態を調査するコマンド

GDB には，デバッグ状態を調査するコマンドとして，"**info**"[1] コマンドがあります。**info** コマンドは，

1)省略形は'i'です。

- **プログラムの状態を調査する**
- **デバッガの状態を調査する**

ために用います。**info** コマンドは，"**info**" と入力した後に，1 つのサブコマンドと (必要ならば) その引数を続けて入力します。

—— ■ 書式 ■ ——

■ info [<サブコマンド>] [<サブコマンドの引数> ...]
デバッガやプログラムの状態を調査する

また，**GDB** Ver. 4 から **info** コマンドと同じような働きをする "**show**"[2] コマンドが追加されました。ただしこのコマンドは，おもに C++ で書かれたプログラムのデバッグ時の設定内容を表示するものです。このコマンドにも **info** コマンドと同じように，1 つのサブコマンドと必要ならばその引数を続けて入力します。

2)省略形は'sho'です。

■書式■

- show [<サブコマンド>] [<サブコマンドの引数> ...]
  おもに C++ プログラム用のコマンド

5.5.2 プログラムの状態を調査するサブコマンド

info コマンドのサブコマンドの中から，デバッグするプログラムの状態を調査するためのサブコマンドについて説明します。

◆ ソースファイルの調査

プログラムがどのソースファイルから作られているのかを調べるときには，“info files”[3]コマンドを使うと便利です。また複数のソースファイルをリンクしたプログラムで，部分的にシンボル情報をつけてコンパイルした場合，どのソースがソースレベルデバッグ可能かを調べるときなどにも便利です。

3)省略形は‘i fi’です。

■書式■

- info files
  ソースファイルを調査する

◆ プログラムに渡された引数の調査

プログラムに渡された引数を調べるには，“show args”[4]コマンドを使います。またプログラムを実行させる場合，そのプログラムに必要な引数を指定し忘れたり，指定した引数を変更したいときは，“set args”[5]コマンドで変更することができます。

4)省略形は‘sho ar’です。

5)省略形は‘set a’です。

■書式■

- show args
  プログラムに渡された <引数> を調査する
- set args <引数> ...
  <引数> を追加／変更する

なお引数を変更した場合，プログラムに新しい引数を渡すために，プログラムを再実行させておきます。なぜならば，プログラムはプログラムを実行するときにだけ，コマンドライン引数を取り込むからです。しかしプログラムに対して同じ引数を与える場合は，プログラムを再スタートさせても，引数を指定する必要はありません。

◆ 関数を調べる

“info functions”[6]コマンドを使用すると，すべての関数名とその関数からの戻り値の型を表示することができます。

6)省略形は‘i fu’です。

また“info functions”と入力した後に続けて文字列を指定することで，指定した文字列にマッチする関数名を，GDB が内部にもっている関数の情報から検索して表示させることもできます。たとえば，関数名を完全に思い出せないときや，名前のつけ方に規則性をもたせている場合に便利です。

■ 書式 ■

- info functions [<文字列>]
  関数名と戻り値の型を表示する

◆ 変数を調べる

GDB は，変数を次に示す 3 つのタイプに分けて管理しています。

❖ グローバル変数／スタティク変数[7]

“info variables”[8]コマンドを使用することで，すべてのグローバル変数とスタティック変数を表示させることができます。同時に，その変数の型も表示されます。また，引数に文字列を指定することにより，“info functions”と同様に，文字列にマッチした変数を検索することができます。

7)実際にメモリ上に割りつけられる変数です。

8)省略形は‘i v’です。

■ 書式 ■

- info variables [<文字列>]
  グローバル変数／スタティック変数をすべて表示する

❖ ローカル変数[9]

“info locals”[10]コマンドを使用することで，実行を停止した位置から参照できるローカル変数を表示することができます。このコマンドは，現在のスコープから参照できるすべてのローカル変数の名称とその値を表示します。

9)関数が呼び出されたときに生成されるスタックフレームに割りつけられます。

10)省略形は‘i lo’です。

■ 書式 ■

- info locals [<文字列>]
  ローカル変数を表示する

11)ローカル変数と同じくスタックフレームに割りつけられます。

12)省略形は'i ar'です。このコマンドは'show args'コマンドとはまったく違うものなので注意してください。

❖ 関数に渡された引数[11)]

"info args"[12)] コマンドを使用することで，関数に渡された引数を表示することができます。

—— ■ 書式 ■ ——

- **info args**
  関数に渡された引数を表示する

---

13)省略形は'i li'です。

14)省略形は'i ad'です。

◆ メモリ上の位置を調べる

引数に指定したソースコードの行番号に対応するメモリ上のプログラムのアドレスを表示するには，"info line"[13)]コマンドを使用します。

また，"info address"[14)]コマンドの引数に変数名または関数名を指定することで，その変数または関数が格納されているアドレスを表示することができます。

—— ■ 書式 ■ ——

- **info line [<行番号>]**
  <行番号> に対応するメモリ上のアドレスを表示する
- **info address <シンボル>**
  <シンボル> に指定した変数 (関数) が格納されているアドレスを表示する

5.6 ························ データを調べる／修正する

このセクションでは，プログラム内の変数を調べる方法と，それを修正する方法について説明します。

5.6.1 変数の内容を表示する

変数の内容を表示するには，“print”[1] コマンドを使います。一般的には，“print” コマンドの後の引数に，内容を表示したい変数名を指定します。またプログラムを書くときと同じ形式で，式を引数にすると，その式を評価して表示します。式は，プログラム中のシンボル[2]と，プログラミング言語の演算子および定数を組み合わせたものです[3]。また，プログラムを記述するときと同様に，指定した変数または定数の型変換を行うことができます。

1)省略形は ‘p’ です。

2)変数名や関数名のことです。

3)print コマンド以外でも引数に値が必要なコマンドは，引数に式を記述することでその値を示すことができます。

―― ■ 書式 ■ ――

■ print [/<出力フォーマット>][<式>]
　変数の内容を表示する

GDB では，プログラミング言語が許可している式に加え，次の 3 種類のオペレータをサポートしています。

- “@”
  メモリを疑似配列として参照するためのバイナリオペレータ
- “::”
  スタティックな関数や変数を外部から参照するためのオペレータ
- “{ 型 } アドレス”
  “アドレス” で指定したメモリを “型” で参照する。たとえば，アドレス 0x6800 を int 型として表示するには print {int}0x6800 と入力する

◆ 疑似配列

メモリ上の同一の型の連続するオブジェクトを表示するために，バイナリオペレータ “@” を使うことができます。つまり，配列を動的に確保したときに，連続する配列の要素を表示するためです。たとえば，

```
int *buff = (int *)malloc (len * sizeof (int));
```

という配列があるとします。このような場合，

```
print *buff @len
```

と入力すると，配列としてすべての要素を表示することができます。

◆ プログラム上の変数

式内の変数は，現在，実行が停止している位置からアクセス可能なローカル変数またはグローバル変数でなければいけません。これは，プログラミング言語のスコープのルールに従っているためです。ただし特例として，メモリ上に割りつけられているスタティック変数またはスタティックな関数の場合は，外部から参照することが許されています。しかしそれでは，同一名称の変数や関数が異なったソースファイルに存在した場合，どれを参照すればよいかわからなくなってしまいます。このようなときは，“::” オペレータを使うことで解決できます。

```
<関数名> :: <変数>
```

または，

```
<関数名> :: <関数>
```

と指定することで，変数または関数を特定することができます。“::” の前の関数名とは，参照したい変数，または関数にアクセス可能な関数のことです。

List 5-2 ● sample1.c

```
1:  static int abc;
2:
3:  foo ()
4:  {
5:    abc = 123;
6:          :
7:  }
```

List 5-3 ● sample2.c

```
1:  foo2 ()
2:  {
3:    static int def = 456;
4:            :
5:            :
6:  }
```

List 5-4 ● sample3.c

```
1:  static int def;
2:
3:  foo3 ()
4:  {
5:    def = 789;
6:          :
7:  }
```

List 5-2 ～ List 5-4 という 3 つのファイルをリンクしたプログラムで説明します。たとえば，現在実行している関数が List 5-2 の **foo ()** 関数であるとします。この場合，C 言語のスコープのルールにしたがえば，スコープが関数内に制限されている List 5-3 の **foo2 ()** 関数の変数 **def**，および スコープがファイル内に制限されている List 5-4 の変数 **def** を参照することはできません。こうしたときに **GDB** の特例を利用して，List 5-3 の変数 **def** にアクセスするには，次のようにします。

```
(gdb) print foo2::def
$1=456
(gdb)
```

また，List 5-4 の変数 **def** の値を表示するには次のようにします。

```
(gdb) print foo3::def
$2=789
(gdb)
```

◆ 出力に関する設定

大きな配列や構造体をわかりやすく表示するために，次に示す 5 つのコマンドが用意されています。

■ 書式 ■

- **set print pretty on**
  構造体を，1 つのメンバが 1 行にインデントされたフォーマットで表示する
- **set print pretty off**
  構造体をコンパクトなフォーマットで表示する (デフォルト)
- **set print union on**
  構造体の中に共用体が含まれている場合に表示する
- **set print union off**
  構造体に含まれる共用体を表示しない

ここで，仮に List 5-5 のような構造体があるとします。

List 5-5 ● struct.c

```
1:  struct {
2:    int idata;
3:    union {
4:      int ival;
5:      float fval;
```

```
6:      char *sval;
7:    } u;
8:  } s;
```

List 5-5 での “set print union on” の出力は，次のようになります。

```
(gdb) set print union on
(gdb) print s;
$1 = {idata=120, u = {ival=1300565, fval=0, sval=0x13d855 "union"}}
(gdb)
```

また，“set print union off” での出力は次のようになります。

```
(gdb) set print union off
(gdb) print s
$1 = {idata = 120, u = {...}}
(gdb)
```

◆ 出力フォーマット

print コマンドなどで変数を表示する場合，評価する変数のデータ型によって表示されるフォーマットが変わります。たとえば，整数型は 10 進数で表示され，ポインタ型は 16 進数で表示されます。しかし，このフォーマットでは都合が悪いこともあります。そのときは出力フォーマットを指定して，表示を変更します。書式は “print” コマンドの後に続けて “/<文字>” という形式です。出力フォーマット指定文字には，次の 6 種類があります。

- “x”　　16 進数で表示する
- “d”　　符号つき 10 進数で表示する
- “u”　　符号なし 10 進数で表示する
- “o”　　8 進数で表示する
- “c”　　文字定数で表示する
- “f”　　浮動小数点表記で表示する

たとえば整数型の val を 16 進数で表示する場合，print/x val と入力します。

5.6.2　メモリの調査

データを評価するさらに低レベルの方法は “x” コマンドを用いることです。このコマンドは，指定されたアドレス上のデータを調査し，指定されたフォーマットで表示します。

■書式■

■ x [/<出力フォーマット>]<アドレス>

<アドレス> で指定したメモリを表示する

また, “x” の後に “/” と「調査サイズ指定文字」,「出力フォーマット指定文字」またはその両方を指定することで, 指定したフォーマットで表示することができます。デフォルトでは, 4 バイトを 16 進数で表示します。また, 調査サイズ指定文字とフォーマット指定文字の前に, 表示するユニット数を指定することができます。たとえば, “x/8h” は, 調査サイズ 2 バイトのユニットを連続して 8 個表示することになります。

引数を省略して “x” と入力することは, 前に “x” コマンドで表示したアドレスの次のアドレスの内容を表示することになります。つまり, ↵ を押していくことで連続してメモリを表示することができます。

“x” コマンドによって表示されるアドレスとその内容は, 変数履歴には代入されません。その代わりに **GDB** では, コンビニエンス変数 “$_” と “$__” の値を式で使うという別の手段を提供しています。 “x” コマンドで最後に表示したアドレスは “$_” に格納され, 表示した内容は, “$__” に格納されます。

調査サイズ指定文字を Table 5-1 に, また, フォーマット指定文字を Table 5-2 に示します。

Table 5-1● 調査サイズ指定文字

指定文字	機　能
b	1 バイト単位で調査する
h	2 バイト単位で調査する
w	3 バイト単位で調査する
g	4 バイト単位で調査する

Table 5-2● 出力フォーマット指定文字

指定文字	機　能
x	符号なし 16 進数で表示する
d	符号つき 10 進数で表示する
u	符号なし 10 進数で表示する
o	符号なし 8 進数で表示する
c	文字定数で表示する[4)]
f	浮動小数点で表示する[5)]
s	null でターミネートされた文字列として表示する[6)]
i	逆アセンブルしてマシンインストラクションを表示する[6)]

4)キャラクタコードとして文字を表示します。

5)これは, ‘w’ と ‘g’ のサイズでのみ有効です。

6)調査サイズ指定文字によって指定されたユニットサイズは無視されます。

たとえば, 0x24050 番地から 4 バイトを 16 進数で表示したい場合は,

```
x/wx 0x24050
```

と入力し, 0x24050 番地から 8 ユニットのマシン命令を表示する場合は,

```
x/8i 0x24050
```

と入力します。

5.6.3 自動表示

通常，プログラムのデバッグは，プログラムを実行しては変数を評価するといったことの繰り返し作業になります。GDB の自動表示機能を使えば，プログラムの実行が停止したときに，自動的に式を評価して表示することができます。この機能を使うには，自動的に表示したい式を“display”[7]コマンドで自動表示リストに加えます。設定が終了すると，指定した式にディスプレイ番号が与えられます。与えられたディスプレイ番号は，自動表示を無効にする場合や削除する場合に指定する番号です。display コマンドには次のものがあります。

7)省略形は‘disp’です。

■書式■

- **display**
  プログラムが停止したときと同様に，自動表示リストに設定された式の値を再表示する
- **display <式>**
  プログラムが停止したときに<式>を自動的に表示するように設定する
- **display /<出力フォーマット><式>** [8]
  print コマンドと同様に，指定したフォーマットで<式>を自動的に表示するように設定する
- **display /<出力フォーマット><アドレス>**
  指定したフォーマットで，<アドレス>からメモリを調査して自動的に表示するように設定する
- **undisplay <ディスプレイ番号>**
  <ディスプレイ番号>で指定した式を自動表示リストから削除する
- **delete display <ディスプレイ番号>**
  <ディスプレイ番号>で指定した式を自動表示リストから削除する
- **disable display <ディスプレイ番号>**
  <ディスプレイ番号>で指定した式の表示を無効にする
- **enable display <ディスプレイ番号>**
  <ディスプレイ番号>で指定した式の表示を有効にする
- **info display**
  自動表示リストに加えた式を表示する。ただし，それぞれの式の値は表示されない

8)「出力フォーマット」を参照（第 5.6.1.4 節 P.200）。

次の List 5-6 は，int 型の配列のデータを昇順にソートするプログラムです。このプログラムを例に display コマンドの使い方を説明します。

List 5-6● シェルソート

```
1:  void shellsort (int v[], int n)
2:  {
```

```
 3:   int gap, i, j, temp;
 4:
 5:   for (gap = n/2; gap > 0; gap /= 2)
 6:     for (i = gap; i < n; i++)
 7:       for (j = i - gap; j >= 0 && v[j] > v[j+gap]; j -= gap)
 8:         {
 9:           temp = v[j];
10:           v[j] = v[j+gap];
11:           v[j+gap] = temp;
12:         }
13: }
```

まずブレークポイントを9行目に設定して，配列 v[] の変化をディスプレイしてみます。“continue” コマンドを実行するたびに，配列 v[] の状態が自動的に表示されていきます。

```
(gdb) break 9
Breakpoint 1 at 0x6a: file ssort.c, line 9.
(gdb) run
Starting program: B:/gdb/ssort.x

Breakpoint 1, shellsort (v=0x19bd32, n=9) at ssort.c:9
9                 temp = v[j];
(gdb) display *v @n
1: *v @ n = {4, 3, 5, 8, 9, 6, 1, 7, 2}
(gdb) c
Continuing.

Breakpoint 1, shellsort (v=0x19bd32, n=9) at ssort.c:9
9                 temp = v[j];
1: *v @ n = {4, 3, 1, 8, 9, 6, 5, 7, 2}
(gdb)
Continuing.

              ⋮

Breakpoint 1, shellsort (v=0x19bd32, n=9) at ssort.c:9
9                 temp = v[j];
1: *v @ n = {1, 2, 3, 4, 6, 5, 7, 8, 9}
(gdb)
Continuing.

Breakpoint 1, shellsort (v=0x19bd32, n=9) at ssort.c:9
9                 temp = v[j];
1: *v @ n = {1, 2, 3, 4, 5, 6, 7, 8, 9}
(gdb)
```

5.6.4 変数履歴

“print” コマンドによって表示された値は，GDB の変数履歴に保存されていきます。そして保存された値には，他の式から参照できるように履歴番号が与えられます。履歴番号は，“print” コマンドで値を表示したときに“$数字 =” という形式で表示されます[9]。

9)この‘$’の次の数字が履歴番号です。

変数履歴から任意の値を参照する場合は，その値の履歴番号を “$履歴番号” という形式で指定します。このとき，単に “$” だけを指定すると，変数履歴に最後に保存された値を参照し，“$$” と指定すると，その直前の値を参照します。また “$$数字” とすれば,「変数履歴に最後に保存された値から何番目」というような指定もできます。

変数履歴を調べるには，次のコマンドを使用します。

―― ■ 書式 ■ ――

- **show values**[10]
  変数履歴の最後から 10 個の値を表示する
- **show values <履歴番号>**
  履歴番号を中心とした 10 個の変数履歴を表示する
- **show values +**[11]
  “print” コマンドで最後に表示した値の後の 10 個の変数履歴を表示する

10)省略形は‘sho va’です。

11)省略形は‘sho va +’です。

たとえば List 5-7 の構造体で，構造体の要素がいくつもリスト構造でつながっている場合，変数履歴を使うことで，リストをたどりながら構造体の内容を表示させていくことが容易にできます。

List 5-7 ● リスト構造体プログラム

```
1:  struct LIST {
2:    char *name;
3:    int val;
4:    struct LIST *next;
5:  };
6:
7:  struct LIST *list;
```

変数 list の差し示す構造体の内容を表示するには，次のように入力します。

```
print *list
```

このコマンドを実行した時点で，変数 list の値が変数履歴に記録されます。次に変数履歴に記録された値を使って，その次の構造体の内容を表示します。

```
print *$.next
```

これで，次の構造体へのポインタが値として変数履歴に記録されます。この後は ↵ を押していくだけで，変数履歴への値の記録と，その値が差し示す構造体の表示を繰り返すことができます。

5.6.5 コンビニエンス変数

コンビニエンス変数は，ユーザが自由に使用することができる便利な変数です。この変数には，“$” で始まる任意の名称をつけることができます。この変数に値を保存するには，プログラムの変数値を設定するときと同様に，式による割り当てを使います。

```
set $foo = 123
```

この例は，コンビニエンス変数 “$foo” を作成し，123 という値を設定した例です。また，すでに作成されている変数に値を設定すると，その変数の値は更新されます[12)]。

たとえば，コンビニエンス変数をインクリメント用のカウンタとした場合，

```
set $i = 0
print table[$i++]->contents
```

と入力した後，↵ を繰り返し押していくことで，“$i” がインクリメントされていきます。

コンビニエンス変数を調べるには，“show convenience” [13)]コマンドを使用します。

■ 書式 ■

- **show convenience**
  設定されているコンビニエンス変数をすべて表示する

12)コンビニエンス変数は，データを記録しておきたい場合に便利です。また，実際の変数のように値に名前がつきますので，他の式から参照することもできます。

13)省略形は ‘sho conv’ です。

5.6.6 プログラム内で C の関数を実行する

C の式からプログラムの中の関数を，直接呼び出すことができます。関数を直接呼び出すには，“call” コマンドを使います。このコマンドは，関数の動作を手早くテストする際に便利です。たとえば関数の呼び出し引数を変えて，繰り返しテストすることによって，その関数からの戻り値が正しいかどうかをチェックすることができます。呼び出し引数には,「定数」,「文字列へのポインタ」または「プログラム内で使われている変数」を渡すことができます。ただし，SX–Window アプリケーションの場合は，引数に文字列を指定しないようにしてください[14)]。

14)その理由は，文字列を引数とした場合，GDB はプログラムの malloc () 関数を使って文字列を格納する領域を確保するからです。

■書式■

- call <式>

  プログラムの関数を直接呼び出す。戻り値が void でなければ，変数履歴に記録される

たとえば2つの引数を指定して gcm () 関数を呼び出した場合，次のようになります。

```
(gdb) call gcm(352,376)
$1 = 8
(gdb)
```

これは，“print” コマンドを使っても同様なことができます。

```
(gdb) print gcm(28,720)
$2 = 4
(gdb)
```

5.6.7　データの変更

変数の値を変更するには，式による割り当てを使う方法と “set” コマンドを使う方法の2通りがあります。

式による割り当てを使う方法では，C言語のすべての割り当て用オペレータを使うことができます。たとえば，

```
print i=4
```

は変数 i に変数 4 という値を格納し，割り当て後の値を表示します。また，

```
print i++
```

は，変数 i をインクリメントします。

割り当て後の値を見る必要がない場合は，“set” コマンドを使います。ただし，このコマンドは，GDB の動作環境を設定するコマンドでもあるため，“set” コマンドに渡す引数の文字列の最初の部分が “set” コマンドのサブコマンド[15]と一致する場合は，“set variable”[16]コマンドを使用する必要があります。このコマンドを使った場合は，式の値が表示されず，変数履歴にも記録されません。たとえば，

```
set i = 0
```

は，変数 i に値変数 0 を格納するだけです。

15)info コマンドのサブコマンドと同じです。

16)省略形は ‘set var’ です。

5.7 ……………………ブレーク／ウォッチポイント

このセクションでは，ブレークポイントとウォッチポイントについて説明します。

5.7.1 ブレークポイント

ブレークポイントは，プログラムの変数やプログラムの状態を調べるために指定した位置でプログラムの実行を停止させる機能です。ブレークポイントを設定する位置には「行番号」,「関数名」,「実際のプログラムのアドレス」が指定できます。また，プログラムが停止するときのさまざまな条件を追加設定することもできます。

GDB では，プログラムの同一箇所での複数のブレークポイントの設定を許可しています。なぜならば，異なった条件のブレークポイントを複数設定することで，さまざまな状況に対応できるからです[1]。

◆ ブレークポイントの設定方法

ブレークポイントは“break”[2]コマンドで設定します。break コマンドには次のようなものがあります。

1)この機能は GDB 以外のデバッガでは許されていないようです。

2)省略形は‘b’です。

―― ■ 書式 ■ ――

- break
  ブレークポイントを現在停止している次の行に設定する
- break [[<ファイル名>:]<行番号>]
  現在選択されているソースファイルの <行番号> で指定した行に，ブレークポイントを設定する。現在選択されているソースファイルとは，複数のソースファイルから 1 つのプログラムが成り立っている場合に，実行を停止した位置が含まれているファイルのことである。つまり，起動する前ならば main () 関数が含まれるファイルということになる。ただし行番号の前にファイルを指定すると，現在選択されているソースファイル以外でも，ブレークポイントを設定することができる
- break [[<ファイル名>:]<関数名>]
  <関数名> で指定した関数のエントリに，ブレークポイントを設定する

- **break ±<オフセット>**
  現在実行を停止している場所からオフセットの行数分だけ，前後にブレークポイントを設定する
- **break *<アドレス>**
  <アドレス> にブレークポイントを設定する
- **break <位置> if <条件>**
  <条件> によって停止するブレークポイントを設定する。<条件> には，設定したブレークポイントにプログラム (の実行) が達したときに評価する式を指定する。また <位置> には，ブレークポイントを設定する「行番号」または「関数名」を指定する
- **tbreak <位置>**[3]
  一時的なブレークポイントを設定する。このブレークポイントは，最初にヒットしたときだけプログラムの実行を停止させ，その後は自動的に削除される
- **rbreak <文字列>**[4]
  引数に指定した文字列にマッチするすべての関数に，ブレークポイントを設定する

◆ ブレークポイントの削除方法

ブレークポイントの削除方法には2通りあります。1つは，ブレークポイントを設定したときの「行番号」,「関数名」または「プログラムのアドレス」を指定して削除する “clear”[5] コマンドを使用する方法です。もう1つは，ブレークポイントを設定したときに表示されるブレークポイントの番号によって削除する “delete”[6] コマンドを使用する方法です。

■ 書式 ■

- **clear [[<ファイル名>:]<関数>]**
  <関数> のエントリに設定されているすべてのブレークポイントを削除する。引数に何も指定しなければ，次に実行される命令に設定されているブレークポイントを削除する
- **clear [<ファイル名>:]<行番号>**
  指定された行のコードに設定されているすべてのブレークポイントを削除する
- **delete [<ブレークポイント番号>]**
  <ブレークポイント番号> に対するブレークポイントを削除する。引数を指定しなければ，すべてのブレークポイントを削除する。ただし，このとき **GDB** は確認を求めてきます。

---

3) 省略形は ‘tb’ です。

4) 省略形は ‘rb’ です。

5) 省略形は ‘cl’ です。

6) 省略形は ‘d’ です。

◆ ブレークポイントの属性

設定済みのブレークポイントが，時には邪魔になることがあります。そのようなときは削除すればよいのですが，後でまた使用することもあります。このようなときのために，ブレークポイントを一時的に無効にすることができます。“enable”[7]と“disable”[8]の両コマンドで，ブレークポイントを有効にしたり，無効にしたりすることができます。ブレークポイントには，実行許可条件に対して次の 4 つの状態があります。

7)省略形は‘en’です。

8)省略形は‘dis’です。

- **許可状態**
  ブレークポイントはプログラムを停止させます。“break” コマンドによって設定されるブレークポイントは，この状態にあります。
- **無効状態**
  ブレークポイントはプログラムに何の影響も与えません。
- **1 回だけ有効な状態**
  ブレークポイントはプログラムを停止させますが，直後に無効状態となります。“tbreak” コマンドによって設定されるブレークポイントは，この状態にあります。
- **削除許可状態**
  ブレークポイントはプログラムを停止させますが，その後，ただちに削除されてしまいます。

またブレークポイントの属性を変更するコマンドには，次のものがあります。

—— ■ 書式 ■ ——

- disable [breakpoints] <ブレークポイント番号>[9]
  指定されたブレークポイントを無効にする
- enable [breakpoints] <ブレークポイント番号>
  指定されたブレークポイントを有効にする
- enable [breakpoints] once <ブレークポイント番号>
  指定されたブレークポイントで，次にプログラムが停止するときまで一時的に有効にする
- enable [breakpoints] delete <ブレークポイント番号>
  指定されたブレークポイントで，次にプログラムが停止するときまで一時的に有効にし，プログラムが停止した後，このブレークポイントを削除する

9)以降，‘[breakpoints]’は省略可能です。

◆ ブレーク条件の変更

ブレークポイントを設定した位置が何度も実行される場合，関心のある状況になるまで実行を停止したくないことがあります。GDB は，ブレークポイントに条件を指定することで，その条件が真のときにだけプログラムの実行を停止させることができます。

ブレーク条件は，“break” コマンドに if <条件> という引数を指定することで設定できます。<条件> には，論理式を記述します。条件つきのブレークポイントは，プログラムがブレークポイントを設定した位置に達したときに条件式を評価します。そして，その条件式が真のときにだけ，プログラムの実行を停止します。また “condition” [10] コマンドを使うことで，設定した条件を変更することができます。 condition コマンドの書式は，次のとおりです。

10)省略形は ‘cond’ です。

──■ 書式 ■──

- condition <ブレークポイント番号><条件>
  <条件> を <ブレークポイント番号> で示すブレークポイントの条件として指定する
- condition <ブレークポイント番号>
  <ブレークポイント番号> で示されるブレークポイントの条件を削除する

たとえば，foo () 関数で変数 x が負のときには実行が停止するようにブレークポイントを設定するには，次のように指定します。

```
break foo if x<0
```

◆ パスカウント

「ある関数を n 回目に呼び出したときにプログラムの実行を停止させたい」とか「ある行を n 回通過したときに停止させたい」という場合のために，ブレークポイントを指定した回数だけ通過したときに，プログラムの実行を停止させる機能があります。つまり，ブレークポイントを無視する回数 (パスカウント) を指定することができます。パスカウントを設定するには，次のコマンドを使用します。

──■ 書式 ■──

- ignore <ブレークポイント番号><カウント>[11]
  <ブレークポイント番号> に示されるブレークポイントに無視する回数を <カウント> に指定する
- count <カウント>[12]
  現在プログラムが停止している位置のブレークポイントに，“<カウント> −1” を設定する

11)省略形は ‘ig’ です。

12)省略形は ‘cou’ です。

もし，同一箇所に条件とパスカウントの両方が設定されている場合は，パスカウントが有効になった後に，条件がチェックされるようになります。

◆ ブレーク時のコマンドの起動

ブレークポイントによってプログラムが停止したとき，ブレークポイントに対して一連のコマンドを起動させることができます。たとえば，他のブレークポイントを無効にしたり，変数の値を表示するなどといったことです。ブレークポイントに対してコマンドを起動する書式は，次のとおりです。

■書式■

■ commands <ブレークポイント番号>[13]

<ブレークポイント番号> に示すブレークポイントによって，プログラムが停止したときに実行するコマンドを指定する。コマンド自身は，続く行に記述し，最後に "end" と入力して，コマンドの入力を終了する

13)省略形は 'com' です。

この機能を無効にするには，"commands" コマンド入力した直後に，"end" を入力します。これは，コマンドに何も与えないことを意味します。

次の例は，1 番のブレークポイントで停止したときに，2 番のブレークポイントを無効にするコマンドを指定したときの画面です。

```
(gdb) commands 1
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
disable 2
end
(gdb)
```

また 3 番のブレークポイントに，プログラムを再スタートさせるコマンドを指定した画面は次のとおりです。

```
(gdb) commands 3
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
echo *** プログラムを再スタートする ***\n
run
end
(gdb)
```

5.7.2 ウォッチポイント

ウォッチポイントは，指定した変数の値の変化を監視し，その値が変化したときにプログラムの実行を停止させる機能です。たとえば，ある変数の値がプログラムが実行するにつれて予想もしない値になるとします。このようなとき，変数にウォッチポイントを設定しておくと，変数がどこで書き換えられているのかを調べることができます。また，変数の値が変化する前と後の値も表示します。

ただし，この機能を使うとプログラムの実行が非常に遅くなります[14]。

14)現在の X68000 では処理が遅すぎて，使いものになりません。

◆ ウォッチポイントの設定

ウォッチポイントの設定には，“watch”コマンドを使います。また,「ウォッチポイントの設定[15]」,「削除[16]」,「停止条件の設定[17]」,「パスカウント[18]」,「コマンドの起動[19]」はブレークポイントと同様に指定できます。 watch コマンドの書式は次のとおりです。

15)「ブレークポイントの設定方法」を参照（第5.7.1節 P.207)。

16)「ブレークポイントの削除方法」を参照（第5.7.2節 P.208)。

17)「ブレーク条件」を参照（第5.7.4節 P.209)。

18)「パスカウント」を参照（第5.7.5節 P.210)。

19)「コマンドの起動」を参照（第5.7.6節 P.210)。

20)省略形は‘wat’です。

―― ■書式■ ――

- watch <シンボル>[20]
  <シンボル> で指定した変数にウォッチポイントを設定する
- watch <アドレス>
  <アドレス> で指定したアドレスにウォッチポイントを設定する

5.7.3 シグナル

GNU オリジナルの **GDB** では，プログラムのプロセスから発生されるシグナルを受ける機能があります。しかし，**Human68k** のシグナルといえば，“CTRL + C” が押されたとき発生する SIGINT くらいしかありません。したがって **Human68k** 版 **GDB** で，シグナル処理の機能を使うのはほとんど無意味なことです。

シグナルに関するコマンドには，次のものがあります。

―― ■書式■ ――

21)省略形は‘i si’です。

22)省略形は‘ha’です。

- info signals[21]
  シグナルの一覧表示をする
- handle <シグナル番号><キーワード> ...[22]
  <シグナル番号> で示したシグナルの処理を変更する
  キーワードには以下のものがある

Table 5-3● シグナル発生時のキーワード

キーワード	機　能
stop	プログラムを停止する
print	メッセージを表示する
nostop	プログラムを停止させない
noprint	メッセージを表示しない

---

23)X680x0 GDB では，LIBC の仕様に合わせて，シグナル名とシグナル番号を変更しています。詳しくは「X680x0 Develop. & libc II」を参照。

また，プログラムのプロセスから受けることができるシグナルには，次の 5 種類があります[23]。

- SIGNMI
  X68000 のインタラプトスイッチが押されたときに発生する

- SIGBUS

  プログラムの実行がバスエラーやアドレスエラーで停止したときに発生する
- SIGILL

  不当命令，0 による除算，CHK 命令，TRAPV 命令によるシグナル
- SIGINT

  CTRL + C が押されたときに発生する
- SIGTRAP

  ブレークポイントによるシグナル

5.8 スタックの調査

関数はコールされるたびに，スタックフレームと呼ばれる関数に関連するデータをもつフレームを生成していきます。このフレームには，関数に与えられた引数，関数のローカル変数が含まれています。そして関数からリターンするときには，その関数のフレームを削除します。したがって，このスタックフレームを調べることで現在実行している関数とその関数が呼び出されるまでの経過を調べることができます。これは，再起呼び出しを使ったプログラムのデバッグに便利です。

5.8.1 バックトレース

バックトレースは，関数が呼び出された経過を調べる機能です。これは，現在実行している関数のフレームから main () 関数のフレームまでさかのぼって表示します。

スタックフレームを調査するには，次のコマンドを使用します。

■書式■

- backtrace[1]
  スタック全体のフレームを 1 行に 1 フレームずつ表示する

1) 省略形は 'bt' です。

backtrace を使用した実行画面は，次のようになります。

```
(gdb) backtrace
#0  term () at test.c:352
#1  0x18b0e6 in expression () at test.c:360
#2  0x18b140 in polish () at test.c:375
#3  0x18acee in keyDownEvent () at test.c:159
#4  0x18ab94 in main () at test.c:64
(gdb)
```

■書式■

- backtrace <フレーム数>[2]

  現在のフレームから <フレーム数> に指定した数だけ表示する。また，<フレーム数> を負にすると，main () 関数のフレームから指定した数だけ表示する

2)省略形は 'bt' です。

5.8.2 フレームの選択

調査したいスタックフレームを選択するには，次のコマンドを使用します。

■書式■

- frame <フレーム番号>[3]

  <フレーム番号> のフレームを選択する。番号は backtrace コマンドで表示されたもの
- frame <アドレス>[4]

  <アドレス> で示されるフレームを選択する
- up <フレーム数>

  <フレーム数>分，上に移動する
- down <フレーム数>

  <フレーム数>分，下に移動する

3)省略形は 'f' です。

4)省略形は 'f' です。

また，フレーム上の情報を表示するには次のコマンドを使用します。

■書式■

- frame[5]

  選択しているスタックフレームの概要を表示する
- info frame[6]

  現在のフレームに関する詳細な情報を表示する
- info frame <アドレス>[7]

  <アドレス> で示したフレームに関する詳細な情報を表示する

5)省略形は 'f' です。

6)省略形は 'i f' です。

7)省略形は 'i f' です。

info frame コマンドを使用した実行画面は，次のようになります。

```
(gdb) info frame
Stack level 0, frame at 0x21425a:
 pc = 0x202b10 in main (t.c:19); saved pc 0x202dba
 source language c.
 Arglist at 0x21425a, args:
 Locals at 0x21425a, Previous frame's sp is 0x214262
 Saved registers:
  fp at 0x21425a, pc at 0x21425e
(gdb)
```

5.9 ソースファイル

このセクションでは，ソースリストの表示およびソースファイルを調査するコマンドを説明します。

5.9.1 ソースファイルの調査

ソースファイルを表示するには，“list”[1] コマンドを使います。

1)省略形は‘l’です。

■ 書式 ■

- **list**
  前に表示したリストの次行から 10 行を表示する。ブレークポイントで停止した後では，停止した位置のまわりの 10 行を表示する
- **list [<ファイル名>:]<行番号>**
  <行番号> で示される行を中心にソースコードを 10 行表示する
- **list [<ファイル名>:]<始まり>,<終わり>**
  <始まり> から <終わり> までの行を表示する
- **list [<ファイル名>:],<終わり>**
  <終わり> が最後の行になるように 10 行表示する
- **list [<ファイル名>:]<関数名>**
  <関数名> の始まりを中心にソースコードを 10 行表示する
- **list +**
  最後に表示した行に続く 10 行を表示する
- **list -**
  最後に表示した行の前の 10 行を表示する

5.9.2 ソースファイルの探索

ソースファイルから指定した文字列を，正規表現を使って検索することができます。文字列の検索には次のコマンドを使用します。

■書式■

- search <文字列>
- forward-search <文字列>
  "list" コマンドで最後に表示した行の次から，指定した <文字列> にマッチするまで後方向に検索し，マッチすればその行を表示する
- reverse-search <文字列>
  "list" コマンドで最後に表示した行の前から，指定した <文字列> にマッチするまで前方向に検索し，マッチすればその行を表示する

5.9.3 ソースファイルのディレクトリ指定

GDB には，ソースファイルを探すためにディレクトリリスト (ソースパス) を表示したり，付け加えたりするコマンド "directory" があります。このコマンドは，GDB がカレントディレクトリ以外にあるソースファイルを必要としたとき，ソースパスに指定されたディレクトリから，必要とするファイル名を見つけるまで探します。directory コマンドには次のものがあります。

■書式■

- directory[2]
  ソースパスを GDB が動作しているカレントディレクトリのみにリセットする。この場合，GDB は確認を求めてくる
- directory <パス> [;<パス> ...][3]
  指定した <パス> をソースパスの最後につけ加える。また，複数のパスを与える場合には，";" で区切る
- show directories[4]
  設定されているソースパスを表示する

2)省略形は 'dir' です。

3)省略形は 'dir' です。

4)省略形は 'sho dir' です。

5.10……………… シンボルテーブルを調査する

シンボルテーブルを直接調査することで，有意義な情報を得ることができます。

5.10.1 シンボルテーブルを調査する

次に示すコマンドで，GDB が読み込んだシンボリックデバッグ情報を調査することができます。

■ 書式 ■

- whatis <シンボル>[1]

  <シンボル> に変数名や関数名を指定すると，その変数または関数の型を表示する。また引数 <シンボル> を省略すると，変数履歴の最後の値の型を表示する

- ptype <型名>[2]

  typedef された <型名> から，そのデータ型に関する詳細を表示する

- info source <ファイル名>

  ソースファイルに関する情報を表示する

- info sources

  現在選択されている[3]ソースファイル名をすべて表示する

- info types[4]

  プログラムに定義されているすべてのデータ型を表示する

- info types <文字列>[5]

  すべてのデータ型のリストから，<文字列> にマッチするデータ型を表示する

- printsyms <ファイル名>[6]

  デバッガが読み込んだシンボリックデバッグ情報を引数に指定した <ファイル> にダンプする[7]

1)省略形は ‘wha’ です。

2)省略形は ‘pt’ です。

3)プログラムの実行が停止した位置に対応するソースコードが含まれているファイルです。

4)省略形は ‘i t’ です。

5)省略形は ‘i t’ です。

6)X680x0 GDB では，廃止されています。

7)<ファイル名> に ‘con’ を指定すると，コンソールに表示することができます。

5.11……………………マシンレベルのデバッグ

ソースレベルデバッグだけでは不十分なときは，マシンレベルでプログラムの実行を制御して，詳しく調査することができます。たとえばプログラムのコーディングミスなどで，コンパイルされた結果が予想もしないマシンコードに変換されていた場合，コンパイルされた「マシンコード」,「CPU レジスタの内容」および「スタックの内容」を調べることによって，バグの原因が明らかになることがあります。

5.11.1 CPU レジスタ

1)第 5.11.2 節 (P.221) を参照。

CPU レジスタを表示するには，“info registers”[1]コマンドを使います。また式からレジスタを参照するには，レジスタ名の頭に“$”をつけます。

CPU レジスタ	GDB でのレジスタ名
d0	$d0
d1	$d1
d2	$d2
d3	$d3
d4	$d4
d5	$d5
d6	$d6
d7	$d7
a0	$a0
a1	$a1
a2	$a2
a3	$a3
a4	$a4
a5	$a5
a6	$fp
a7	$sp
ステータスレジスタ	$ps
プログラムカウンタ	$pc

Fig. 5-1● CPU レジスタと GDB でのレジスタ名の対応

たとえばプログラムカウンタを 16 進数で表示する[2]には，次のように入力します。

2)第5.6.1節(P.197)を参照。

```
print/x $pc
```

またプログラムカウンタが示すアドレスのマシン命令を表示したいならば[3]，次のようになります。

3)第5.6.1節(P.197)を参照。

```
print/i $pc
```

最後はスタックポインタに 4 を加える[4]方法です。

4)第5.6.7節(P.206)を参照。

```
set $sp += 4
```

5.11.2 レジスタを表示するコマンド

68000CPU のレジスタの値を表示するには，次のコマンドを使用します。

■ 書式 ■

- **info registers**[5]
  すべてのレジスタの名称とレジスタの内容を表示する
- **info registers <レジスタ名>**[6]
  指定したレジスタの内容を表示する。<レジスタ名> は頭に “$” のついた名称である

5)省略形は ‘i reg’ です。なお X680x0 GDB では，表示形式が変更されています。

6)X680x0 GDB では，表示形式が変更されています。

5.11.3 マシンレベルのステップ実行

マシンレベルでステップ実行をするには，次のコマンドを使用します。

■ 書式 ■

- **stepi [<リピート回数>]**[7]
  マシン命令単位でステップ実行する。サブルーチン呼び出しがある場合は，そのサブルーチンの中に入る
- **nexti [<リピート回数>]**[8]
  マシン命令単位でステップ実行する。サブルーチン呼び出しがある場合は，サブルーチンを一度に実行する

7)省略形は ‘si’ です。

8)省略形は ‘ni’ です。

5.11.4 逆アセンブル

コンパイラが，ソースコードをどのようなマシン命令に変換したのかを“**disassemble**”コマンドで，逆アセンブルして調べることができます。また，このコマンドで表示される逆アセンブルリストは，**Human68k** の DOS コールや SX-Window のシステムコールを，そのコールの名称で表示します。逆アセンブルコマンドには次のものがあります。

──■書式■──

- **disassemble**
  現在の位置から表示する
- **disassemble <開始>[<終了>]**[9]
  <開始> から **<終了>** まで逆アセンブルして表示する
- **disassemble <関数名>**
  指定した関数のエントリからその関数の終わりまで，逆アセンブルして表示する

9)省略形は‘disas’です。

disassemble コマンドを使用した実行画面は，次のようになります。

```
(gdb) disassemble Init
Dump of assembler code for function Init:
to 0x18af4e:
0x18ae4a <Init>:        link.w fp,#-512
0x18ae4e <Init+4>:      pea 0x0
0x18ae52 <Init+8>:      pea 0x18af4e <EndPr>
0x18ae58 <Init+14>:     jsr 0x18aa5a <TSSetAbort>
0x18ae5e <Init+20>:     __TSGetID
0x18ae60 <Init+22>:     move.l d0,-30834(a5)
0x18ae64 <Init+26>:     move.w #-1,-(sp)
0x18ae68 <Init+30>:     pea -512(fp)
0x18ae6c <Init+34>:     __TSGetTdb
0x18ae6e <Init+36>:     __SXVer
0x18ae70 <Init+38>:     movea.l d0,a0
0x18ae72 <Init+40>:     movea.w a0,a0
0x18ae74 <Init+42>:     move.l a0,-30830(a5)
0x18ae78 <Init+46>:     lea 14(sp),sp
0x18ae7c <Init+50>:     cmpa.l #257,a0
(gdb)
```

5.11.5 マシンレベルデバッグの例

“**display**”[10]コマンドを使うと，プログラムの実行が停止したとき，次に実行

10)第 5.6.3 節 (P.202) を参照。

するマシン命令を逆アセンブルして自動的に表示させることができます。実行画面は次のようになります。

```
(gdb) display/i $pc

1: x/i $pc  0x18ae4e <Init+4>: pea 0x0
(gdb) stepi
0x18ae52 255   TSSetAbort (&EndPr, 0); /* アボート処理ルーチン登録 */
1: x/i $pc  0x18ae52 <Init+8>: pea 0x18af4e <EndPr>
(gdb)
0x18ae58 255   TSSetAbort (&EndPr, 0); /* アボート処理ルーチン登録 */
1: x/i $pc  0x18ae58 <Init+14>: jsr 0x18aa5a <TSSetAbort>
(gdb)
0x18aa5a in TSSetAbort ()
1: x/i $pc  0x18aa5a <TSSetAbort>: move.l -32710(a5),d0
(gdb)
```

次の画面は，CPU レジスタの内容をコンパクトに表示するユーザコマンドの作成例です。

```
(gdb) define ar
Type commands for definition of "ar".
End with a line saying just "end".
printf "SR = %04X", $ps
printf "D %08X %08X %08X %08X\n  %08X %08X %08X %08X\n",
          $d0, $d1, $d2, $d3, $d4, $d5, $d6, $d7
printf "A %08X %08X %08X %08X\n  %08X %08X %08X %08X\n",
          $a0, $a1, $a2, $a3, $a4, $a5, $fp, $sp
end
(gdb)
```

このコマンドの実行結果は，次のようになります。

```
(gdb) ar
SR = 0004
D 00010201 0018AF4E 00000001 00000000
  00000000 00000000 00000000 00000000
A 00000201 0018AAAE 0018AAA6 000A18B4
  00000000 0020CF72 0020B0AC 0020AE9A
(gdb)
```

5.12……………………………… コマンドシーケンス

このセクションでは，ユーザ定義コマンドの設定方法を説明します。

5.12.1　ユーザ定義コマンド

ユーザ定義コマンドとは，GDB に用意されているコマンドを使ってユーザが新たに作るコマンドのことです。たとえば，一連のコマンド実行を 1 つのコマンドに割りつけることができます。

──■書式■──

■ define <コマンド名>[1)]

<コマンド名> という名称で新しいコマンドを定義する。もし，すでに同名のコマンドが存在する場合は，それを再定義するのかどうかの確認を求めてくる

1)省略形は 'def' です。

ただし，ユーザ定義コマンドには制限があります。次のことに注意してください。

- **引数を受けつけない**
- **任意のエラーはユーザ定義コマンドの実行を停止させる**
- **確認を求めてくるコマンドは確認を求めなくなる**
- **コマンドが表示するメッセージは省略される**
- **コマンド名の省略形はビルトインコマンドが優先される**

次の画面は，設定してあるブレークポイントの中から特定のブレークポイントだけを一度に無効にし，実行再開するコマンドです。

```
(gdb) define mukou
Type commands for definition of "mukou".
End with a line saying just "end".
disable 2
disable 4
disable 5
cont
end
(gdb)
```

また下の画面のように，GDB のコマンド名が気にいらなければ，別名定義コマンドとして利用することもできます。

```
(gdb) define start
run
end
(gdb)
```

5.12.2 ドキュメンテーション

“define” コマンドで作成したユーザ定義コマンドに説明[2]を設定する (ドキュメンテーション) ことができます。ユーザ定義コマンドへのドキュメンテーションには，次のコマンドを使用します。

2)help コマンドで表示されます。

■書式■

- document <コマンド名>[3]
  ユーザ定義の <コマンド名> にドキュメンテントがつけられる。ただし，<コマンド名> のコマンドはすでに定義されている必要がある。設定したドキュメントは，“help” コマンドを実行したときに表示される

3)省略形は ‘doc’ です。

たとえば下の画面は，ユーザ定義コマンド “start” にドキュメントをつけたものです。

```
(gdb) document start
プログラムの実行を開始します。
end
(gdb)
```

5.12.3 コマンドファイル

コマンドファイルとは，**GDB** のコマンドを行としてもつファイルのことです。つまり，DOS のバッチファイルのようなものと考えればよいでしょう。

GDB が起動したときに，**GDB** の初期化ファイルである“`.gdbinit`”というファイルを自動的に実行します。これは，DOS でいう `AUTOEXEC.BAT` ファイルのことです。**GDB** は初期化ファイルを環境変数“`HOME`”で指定されたホームディレクトリや，カレントディレクトリ[4]からも読み込みます。ただし，**GDB** 起動時に“`-nx`”オプションが指定されていた場合は，初期化ファイルは実行されません。また，“`source`”コマンドで，コマンドファイルの実行をリクエストすることができます。

4)カレントディレクトリに存在している場合です。

次のように，コマンドファイルの書式は1行に1つのコマンドを並べていくだけです。そして，コマンドファイルをコメント行にするには，行の先頭に“#”を記述します。

―― ■書式■ ――

- `source` <ファイル名>[5]

  <ファイル名> というコマンドファイルを実行する

5)省略形は‘so’です。

5.12.4 出力調整用のコマンド

ユーザ定義コマンドやコマンドファイルの中で，メッセージを表示したいときに便利なコマンドです。

―― ■書式■ ――

- `echo` <文字列>[6]

  <文字列> を表示する
- `output` <式>[7]

  `print` コマンドと同様に <式> の値を表示するが，“`$n = `”は表示しない。また改行もしない。そして，値は変数履歴に記録されない
- `output /`<出力フォーマット><式>[8]

  <出力フォーマット> に従って <式> の値を表示する
- `printf "`<フォーマット文字列>`",`<式>`, ...`

  C 言語の `printf ()` 関数のように，<フォーマット文字列> に従って表示する

6)省略形は‘ec’です。

7)省略形は‘ou’です。

8)<出力フォーマット>については第5.6.1.4節(P.200)の「出力フォーマット」を参照。

`printf` コマンドは，次のように C 言語と同様の形式で記述することができます。

```
printf "変数 x の値は %d です。\n", x
```

5.13 カスタマイズ

このセクションでは，GDB の動作環境の設定と GDB を制御するコマンドについて説明します。

5.13.1 GDB のコマンドシェル機能

GDB には次のような簡単なコマンドシェルの機能があります。

◆ プログラムのパス

GDB では，デバッグ対象のプログラムがワーキングディレクトリに存在しない場合は，環境変数 “PATH” 設定されたディレクトリのリストに従ってファイルを探します。また GDB の “path” コマンドを使用することで，パスリスト を追加することができます。

■ 書式 ■

- **path**
  実行ファイル検索リストを表示する
- **path <パスリスト>**
  引数に指定したディレクトリを実行ファイル検索リストに追加する

5.13.2 プログラムの環境

プログラムは，GDB から環境変数をそのまま受け継ぎます。しかし，“set environment” コマンドまたは “unset environment” コマンドを使えば，デバッガから抜け出すことなく環境変数の一部を変更することができます。

環境変数を変更するコマンドには，次のものがあります。

—■書式■—

- **set environment <変数名> = <値>**[1]
  環境変数に値を設定する
- **unset environment <変数名>**[2]
  環境変数の中から指定した変数を削除する
- **show environment**[3]
  環境変数をすべて表示する
- **show environment <変数名>**
  指定した環境変数を表示する

1)省略形は 'set en' です。

2)省略形は 'uns en' です。

3)省略形は 'sho en' です。

5.13.3 ワーキングディレクトリ

通常 **GDB** のワーキングディレクトリは，**GDB** を起動したディレクトリになります。しかし，**GDB** の起動時オプション "-cd=DIR" または "cd" コマンドを使うことで，ワーキングディレクトリを変更することができます。

—■書式■—

- **cd <パス>**
  GDB のワーキングディレクトリを引数に指定したディレクトリに変更する
- **pwd**
  GDB のワーキングディレクトリを表示する

5.13.4 プログラムへの入出力

プログラムは，デフォルトでは **GDB** と同一の標準入出力が割り当てられます。仮にプログラムに対して標準入出力を変更したいならば，プログラムの実行を開始させる "run" [4]コマンドを使うことで，DOS のコマンドシェルと同様な方法でリダイレクトを行うことができます。

4)「プログラムの実行」を参照(第5.4.1節 P.189)。

—■書式■—

- **run >〔ファイル名〕**
  標準出力を〔ファイル名〕に出力する
- **run <〔ファイル名〕**
  〔ファイル名〕から標準入力へ読み込む

- **run >>** 〔ファイル名〕
  標準出力を〔ファイル名〕にアペンドする
- **run >&** 〔ファイル名〕
  標準エラー出力を〔ファイル名〕に出力する

また標準入出力を変更する別の方法として，“tty” コマンドを使う方法があります。このコマンドの場合は，標準入出力のすべてが “tty” で指定したデバイスに割りつけられます。たとえば，プログラムの入出力を AUX デバイスから行いたい場合に有効です。

—— ■ 書式 ■ ——

- **tty <デバイス名>**[5]
  引数に指定した <デバイス> に標準入出力を割りつける

5)省略形は ‘tt’ です。

ここで説明したリダイレクト機能は，プログラムに対してだけ働くもので，**GDB** の標準入出力には影響しません。

5.13.5 コンソールの切り替え

GDB の標準入出力は，通常 **X68000** のコンソールから行われますが，“remote” コマンドを使うことで変更することができます。たとえば，RS–232C で接続したターミナルがある場合，**GDB** の操作を **X68000** とターミナルの間で切り替えることができます。

—— ■ 書式 ■ ——

- **remote aux**
  GDB の標準入出力を AUX(RS–232C で接続されたターミナル) に切り替える
- **remote con**
  GDB の標準入出力を CON(X68000) に切り替える

5.13.6 画面の切り替え

スクリーンスワップモードで **GDB** を起動した場合は，“screen” コマンドで **GDB** の画面とプログラムの画面とを切り替えることができます。このコマンドは，スクリーンスワップモードで起動した場合にのみ有効です。

──■書式■──

■ screen
GDBの画面からプログラムの画面へ切り替える。画面をGDBに戻すには ↵ を押す

■ screen <背景の色>,<文字の色>
引数にGDBの画面の表示色を0 ~65535[6]までのカラーコードで設定する

6)16進数で指定する場合は数値の前に'0x'をつけます。

5.13.7 チャイルドプロセスの起動

GDBからは，チャイルドプロセスを起動することもできます。

──■書式■──

■ shell <コマンド>[7]
<コマンド>に指定したコマンドを，チャイルドプロセスとして起動する。引数に何も指定しなければ，環境変数"SHELL"に従ってシェルを起動する

7)省略形は'she'です。

5.13.8 プロンプト

GDBでは，プロンプト[8]が表示されていればコマンドの入力が可能です。このプロンプトは，"set prompt"コマンドで自由に変更することができます。

8)デフォルトでは'(gdb)'。

──■書式■──

■ set prompt <プロンプト>[9]
GDBのプロンプトを引数に指定した新しい<プロンプト>に変更する

9)省略形は'set pro'です。

たとえば，

```
set prompt (top-gdb)
```

は，プロンプトを"(top-gdb)"に変更するコマンドです。GDBをGDBでデバッグするときなど，上記のように変更することで，どちらのGDBのプロンプトか判断しやすくなるメリットがあります。

■ 書式 ■

- **show prompt**[10]
  現在のプロンプトを表示する

10)省略形は 'sho pro' です。

5.13.9 コマンド行編集

通常，GDB のコマンド入力は，GDB にリンクされている GNU readline によって行われます。GNU readline は，エディタ感覚で行編集が行えるコマンドラインインタフェイスです。特徴は Emacs スタイル，または vi スタイルのキー操作 (Vol. 2，P.211 参照) が可能で，HISTORY.X のようなヒストリ機能も使えることなどです。この行編集の機能を使うには "set editing" コマンドで設定します。デフォルトでは，行編集を行えるようになっています。

■ 書式 ■

- **set editing [on]**[11]
  コマンドを入力する際に，行編集が行えるように設定する
- **set editing off**
  コマンドを入力する際に，行編集が行えないように設定する
- **show editing**[12]
  行編集機能の設定状態を表示する

11)省略形は 'set ed' です。

12)省略形は 'sho ed' です。

5.13.10 コマンドヒストリ

コマンドヒストリの設定には，"set history" コマンドを使います。

■ 書式 ■

- **set history save [on]**[13]
  記録されたコマンドヒストリを，"set history filename" コマンドで設定したファイルに書き出す
- **set history save off**
  コマンドヒストリをファイルに書き出さないようにする
- **set history size <サイズ>**[14]
  ヒストリバッファのサイズを設定する

13)省略形は 'set hi sa' です。

14)省略形は 'set hi si' です。

- **set history filename <ファイル名>** [15]
  指定した <ファイル名> をヒストリファイル名に設定する。デフォルトでは，"./.gdb_history"[16]
- **show history**[17]
  ヒストリ機能の状態を表示する
- **show history filename**[18]
  ヒストリバッファを書き出すファイル名を表示する
- **show history save**[19]
  ヒストリバッファをファイルに書き出す機能の状態を表示する
- **show history size**[20]
  ヒストリバッファのサイズを表示する
- **show commands**[21]
  記録されているコマンドヒストリから最後の 10 個を表示する
- **show commands <ヒストリ番号>**[22]
  ヒストリ番号を中心として，10 個のコマンドを表示する

15)省略形は 'set hi f' です。

16)Ext 氏の Twenty One が常駐していなければ '._gdb_history' です。

17)省略形は 'sho hi' です。

18)省略形は 'sho hi f' です。

19)省略形は 'sho hi sa' です。

20)省略形は 'sho hi si' です。

21)省略形は 'sho com' です。

22)省略形は 'sho com' です。

5.13.11 スクリーンサイズの設定

画面表示を見やすくするために，画面に合わせてスクリーンサイズを設定します。

■書式■

- **set height <行数>**[23]
  使用するコンソールの行数を設定する。**X68000** のコンソールを使用する場合は自動的に 31 行に設定されるため，リモートコンソールモードを使用する場合に接続したターミナルの画面の大きさに合わせて設定するとよい
- **set width <桁数>**[24]
  使用するコンソールの桁数を設定する。**X68000** のコンソールを使用する場合は，自動的に 96 桁に設定される
- **show height**[25]
  スクリーンの行数を表示する
- **show width**[26]
  スクリーンの桁数を表示する
- **set listsize <行数>**[27]
  "list" コマンドで一度に表示される行数を引数で設定する

23)省略形は 'set he' です。

24)省略形は 'set wi' です。

25)省略形は 'sho he' です。

26)省略形は 'sho wi' です。

27)省略形は 'set li' です。

5.13.12 デフォルト数値の設定

数値を入力または表示する場合，何進数で入力／表示するのかについて設定します。

―― ■ 書式 ■ ――

- set radix <ベース>[28)]
  数値を入力または表示する場合のデフォルトの基数を設定する。引数の <ベース> に与える値は，2，8，10，16 のいずれかになる
- show radix[29)]
  現在の基数を表示する

28)省略形は 'set r' です。

29)省略形は 'sho r' です。

5.13.13 GDB からのワーニングとメッセージ

GDB は，ユーザにとって興味があると思われる **GDB** の動作情報を表示することができます。これらのうちのいくつかは，"`set verbose`" コマンドで切り替えることができます。

―― ■ 書式 ■ ――

- set verbose on[30)]
  GDB のいくつかの情報メッセージを許可する
- set verbose off[31)]
  GDB のいくつかの情報メッセージを抑止する
- show verbose[32)]
  "`set verbose`" コマンドによって設定された現在の状態を表示する
- set confirm on[33)]
  確認を求めるように設定する
- set confirm off[34)]
  GDB は，いくつかのコマンドで確認を求めることがあるが，このコマンドによって確認を求めないように設定することができる
- show confirm[35)]
  "`confirm`" の現在の状態を表示する

30)省略形は 'set ve on' です。

31)省略形は 'set ve off' です。

32)省略形は 'sho ve' です。

33)省略形は 'set con on' です。

34)省略形は 'set con off' です。

35)省略形は 'sho con' です。

5.13.14　シンボルテーブルに関しての設定

もし，プログラムのデバッグ情報にバグがある場合などは，“set complaints”コマンドでいくつかのエラーを無視することができます。

―― ■書式■ ――

- set complaints <無視する回数>[36)]
  シンボルテーブルのエラーを無視する回数を設定する
- show complaints[37)]
  現在設定されているエラーを無視する回数を表示する
- set symbol-reloading on
  プログラムを再スタートする場合，シンボルテーブルを読み込むように設定する
- set symbol-reloading off
  プログラムを再スタートする場合，シンボルテーブルを読み込まないように設定する
- show symbol-reloading
  “set symbol-reloading” コマンドで設定された状態を表示する

36)省略形は ‘set com’ です。

37)省略形は ‘sho com’ です。

5.14……………………………………GDB の制限

Human68k 版 **GDB** には，その仕様によるいくつかの制限があります。

❖ **デバッグ可能プログラムサイズ**

GDB が起動した後のフリーメモリに依存します。ただし，**Human68k** 版 **GDB** は，シンボル情報をすべて読み込みメモリに格納するため，シンボル情報のサイズだけフリーメモリが少なくなります。

❖ **シンボル**

シンボルテーブルのサイズ，シンボル名の長さは無制限ですが，フリーメモリに依存します。

❖ **コマンドライン**

プログラムに渡すコマンドラインには，ワイルドカードや環境変数を使った指定ができません。なぜならば **Human68k** 版 **GDB** では，**GDB** から直接プログラムを起動するため，シェル機能を使うことができないからです。しかしオリジナルの **GDB** では，**GDB** からシェルをチャイルドプロセスとして起動し，そのシェルからプログラムを起動するようになっています。

❖ **シグナル**

オリジナルの **GDB** では，チャイルドプロセス[1]からのシグナルを受けることで，プログラムの状態を調査することができます。しかし **Human68k** 版 **GDB** では，OS の仕様上ほとんど意味のない機能です。

1)デバッグ対象プログラム。

❖ **キーボード入力**

SX-Window 上で動作するプログラムのデバッグ中，キーボードからの入力を受けつけない状態になることがあります。このような場合には，以下の方法を試してください。

- CTRL + OPT.1 + F10 を押す
- マウスを動かしてみる

Chapter 6

Appendix A

6.1 ……………………… 各ファイルのフォーマット

このセクションでは，オブジェクトファイルのコマンドとフォーマット，ライブラリファイルのフォーマット，実行ファイルのフォーマットについて説明します。

6.1.1　オブジェクトファイルのフォーマット

ここでは，オブジェクトファイルのフォーマットを説明します。オブジェクトファイルは，オブジェクトファイルのコマンド，シンボリックデバッグ情報から構成されていて，Fig. 6-1のような順番で並んでいます。シンボリックデバッグ情報はなくてもかまいません。

オブジェクトファイルの情報
各セクションの情報
（指定されていれば）プログラムの実行アドレス， オブジェクトファイルのリンク要求
シンボルの定義
上記以外のコマンド
終了コマンド
シンボリックデバッグ情報

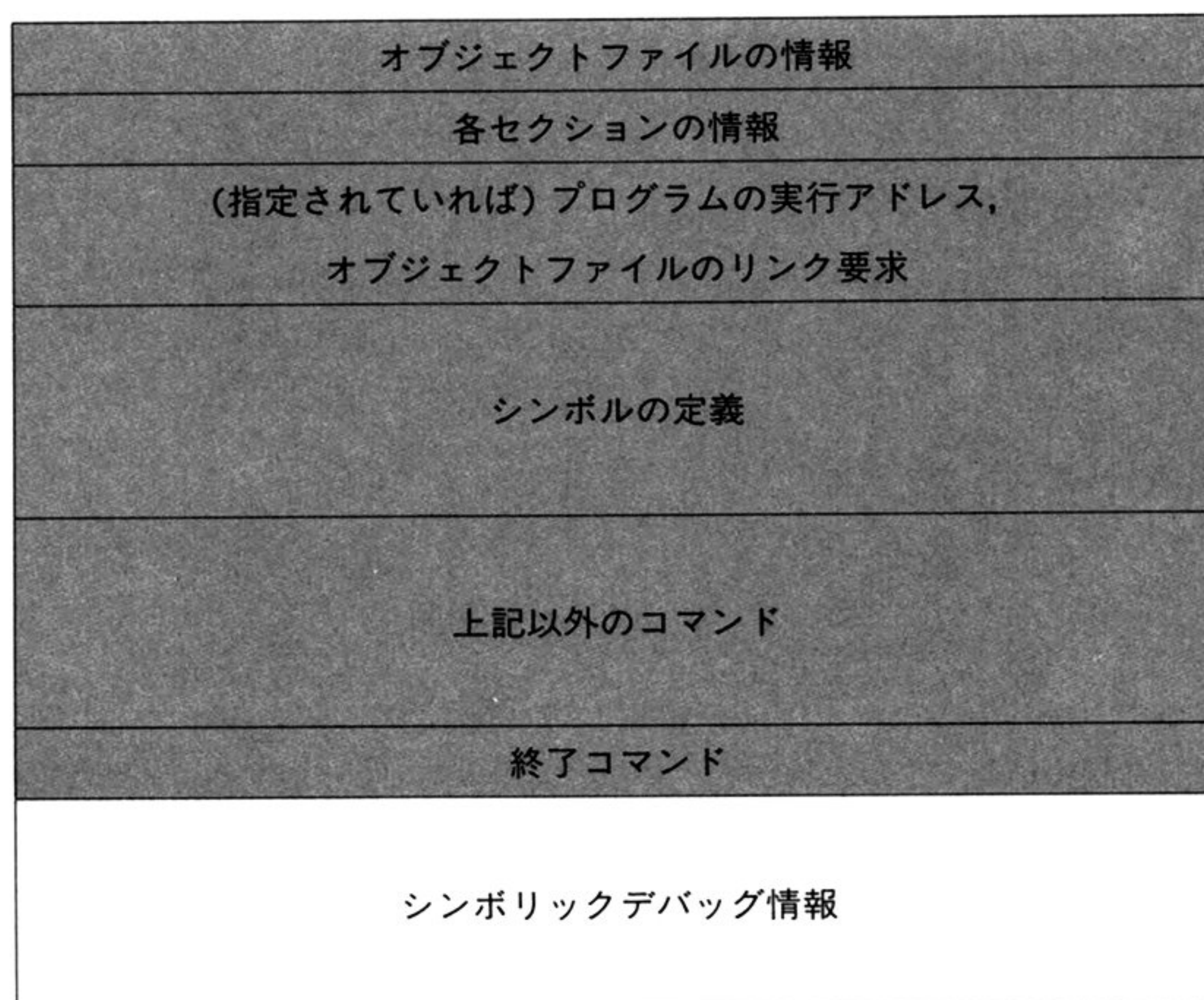



Fig. 6-1● オブジェクトファイルのフォーマット

6.1.2 オブジェクトファイルのコマンド

ここでは，オブジェクトファイルのコマンドについて説明します。

オブジェクトコマンドには，いくつかのシンボルの意味と値を表すための独自の表現方法があります。これらについて説明をしましょう。

❖ SECTION

セクションの番号を表します。番号とセクションとの対応は Table 6-1 のとおりです。

Table 6-1● セクションの番号とセクション名との対応

番号（16 進数）	セクション	
01	テキストセクション	
02	データセクション	
03	ブロックストレージセクション	
04	スタックセクション	
05	相対データセクション	(64K バイト以内)
06	相対ブロックストレージセクション	(64K バイト以内)
07	相対スタックセクション	(64K バイト以内)
08	相対データセクション	(64K バイト以上)
09	相対ブロックストレージセクション	(64K バイト以上)
0a	相対データセクション	(64K バイト以上)

❖ SECTION:ADDRESS

あるセクション (SECTON) でのアドレスの値 (ADDRESS) を実行ファイルの先頭からのアドレスに変換した値です。たとえばデータセクションが 1648 番地から始まる場合，02:000042 はデータセクションの先頭から 42 番地目になるので，変換した値は 168a になります。

❖ XREF_LABEL_SECTION

外部参照シンボルがコモンエリアのシンボルか，普通の外部参照シンボルかを示す番号のことです。番号と属性[1)]の対応は，Table 6-2 のとおりです。

1)「属性」を参照 (第 4.5.1 節 P.155)。

Table 6-2● 番号と属性の対応

番号（16 進数）	属　性	
ff	外部参照シンボル	
fe	コモンエリアのシンボル	
fd	相対コモンエリアのシンボル	(64K バイト以内)
fc	相対コモンエリアのシンボル	(64K バイト以上)

❖ LABEL_NO

外部参照シンボルは各オブジェクトファイルごとに，出現順に 0 番から 1, 2, 3 と番号が割り当てられます。LABEL_NO はこれらの番号を表します。

❖ 演算用スタック

外部参照シンボルどうしの演算は，すべて逆ポーランド方式で行われます。そして 演算用スタックは，これらの演算を行うためのワークエリアを指します。このワークエリアは，ロングワードサイズの値と属性をペアで格納する

ことができます。また，このスタックは1024個分の大きさがあるので，普通に使用するならばオーバーフローすることはありません。このスタックポインタのことをCSP(Calc Stack Pointer)と呼ぶことにします。

6.1.3　オブジェクトファイルのコマンド一覧

オブジェクトコマンドは，大別すると「定義」,「演算」,「制御」,「書き込み」という4つのセクションに分類できます。ここでは，各セクションごとにまとめて説明することにします。

コードの項目には，コマンドとそのフォーマットが示されています。フォーマットはアセンブラ形式で書いてありますが，アセンブラプログラムではなく，データのサイズと並び方を表しています。最初の1バイトがコマンドのコードになります。

機能の項目では，そのコードがどのような働きをするのかについて説明します。

◆ 定　義

外部参照または外部定義シンボルの登録，コモンエリアの登録，各セクションのサイズの定義等を行います。

コード
```
.dc.b $b2,$00
.dc.l VALUE
.dc.b "LABEL_NAME"
.dc.b 0
.even
```

機　能　値がVALUEで，属性が絶対値の外部参照シンボル“LABEL_NAME”を登録します。

コード
```
.dc.b $b2,SECTION
.dc.l ADDRESS
.dc.b "LABEL_NAME"
.dc.b 0
.even
```

機　能　値がSECTION:ADDRESSで，属性がアドレスの外部参照シンボル“LABEL_NAME”を登録します。

コード
```
.dc.b $b2,$fe
.dc.l ADDRESS
.dc.b "LABEL_NAME"
.dc.b 0
.even
```

機　能　サイズが SIZE の相対コモンエリア (64K バイト以内) のシンボル "LABEL_NAME" を登録します。

コード
```
.dc.b $b2,$fd
.dc.l ADDRESS
.dc.b "LABEL_NAME"
.dc.b 0
.even
```

機　能　サイズが SIZE の相対コモンエリア (64K バイト以上) のシンボル "LABEL_NAME" を登録します。

コード
```
.dc.b $b2,$fc
.dc.l ADDRESS
.dc.b "LABEL_NAME"
.dc.b 0
.even
```

機　能　サイズが SIZE のコモンエリアのシンボル "LABEL_NAME" を登録します。

コード
```
.dc.b {$b0, $b2},$ff
.dc.l LABEL_NO
.dc.b "LABEL_NAME"
.dc.b 0
.even
```

機　能　外部参照シンボル "LABEL_NAME" を登録します。

コード
```
.dc.b $c0,SECTION
.dc.l SECTION_SIZE
.dc.b "SECTION_NAME"
.dc.b 0
.even
```

機　能　セクションの情報です。セクションのサイズ情報がわかります。

コード
```
.dc.b $d0,$00
.dc.l FILE_SIZE
.dc.b "FILE_NAME"
.dc.b 0
.even
```

機　能　オブジェクトファイルの情報です。ファイル名とサイズ情報がわかります。サイズ情報は シンボリックデバッグ情報のサイズを含みません。

コード
```
.dc.b $e0,$00
.dc.w SECTION
.dc.l ADDRESS
```

機　能　実行ファイルの実行アドレスを SECTION:ADDRESS に設定します。複数のオブジェクトファイルで指定された場合は，最後に指定したオブジェクトファイルのアドレスが有効になります。

◆ 演　算

外部参照シンボルの値を使った演算を行います。一部の簡単な演算は書き込みコマンドに用意されています。

コード
```
.dc.b $80,XREF_LABEL_SECTION
.dc.w LABEL_NO
```

機　能　外部参照のシンボルの値と属性を演算用スタックに積みます。

コード
```
.dc.b $80,SECTION_NO
.dc.l ADDRESS
```

機　能　SECTION:ADDRESS の値と属性を演算用スタックに積みます。

コード　`.dc.b $93,$00`

機　能　CSPの先頭から値を取り出して，バイトサイズの値とみなして書き込みます。
書き込む値が `-$80` ～ `$ff` の範囲に入っていない場合は，Over flow エラーになります。書き込む値の属性がアドレスを示す場合は，Relative エラーになります。

コード　`.dc.b $91,$00`

機　能　CSP の先頭から値を取り出して，ワードサイズの値とみなして書き込みます。
書き込む値が `-$8000` ～ `$ffff` の範囲に入っていない場合は，Over flow エラーになります。書き込む値の属性がアドレスの場合は，Relative エラーになります。

コード　`.dc.b {$92, $96},$00`

機　能　CSP の先頭から値を取り出して，ロングワードサイズの値とみなして書き込みます。

コード　`.dc.b $90,$00`

機　能　CSP の先頭から値を取り出して，ワードサイズの値とみなして書き込みます。
書き込む値が `-$80` ～ `$ff` の範囲に入っていない場合は，Over flow エラーになります。書き込む値の属性がアドレスの場合は，Relative エラーになります。

コード　`.dc.b $99,$00`

機　能　CSP の先頭から値を取り出して，ワードサイズの値とみなして書き込みます。
書き込む値が，`-$8000` ～ `$7fff` の範囲に入っていない場合は，Over flow エラーになります。書き込む値の属性がアドレスの場合は，Relative エラーになります。

スタック演算コマンド

スタック演算コマンドを Table 6-3 に示します。比較演算子の場合，真ならば −1 を，偽ならば 0 を返します。それ以外の演算子では演算結果をスタックに積みます。

演算子は，アセンブラと同じ演算結果を返します。

Table 6-3● スタック演算コマンド

コード	演算
$a0,$01	(CSP) = −(CSP)
$a0,$02	(CSP) = (CSP)
$a0,$03	(CSP) = *.not.* (CSP)
$a0,$04	(CSP) = *.high.* (CSP)
$a0,$05	(CSP) = *.low.* (CSP)
$a0,$06	(CSP) = *.highw.* (CSP)
$a0,$07	(CSP) = *.loww.* (CSP)
$a0,$08	未定義
$a0,$09	(CSP + 1) = (CSP + 1) * (CSP); CSP = CSP + 1
$a0,$0a	(CSP + 1) = (CSP + 1) / (CSP); CSP = CSP + 1
$a0,$0b	(CSP + 1) = (CSP + 1) % (CSP); CSP = CSP + 1
$a0,$0c	(CSP + 1) = (CSP + 1) .shr. (CSP); CSP = CSP + 1
$a0,$0d	(CSP + 1) = (CSP + 1) .shl. (CSP); CSP = CSP + 1
$a0,$0e	(CSP + 1) = (CSP + 1) .asr. (CSP); CSP = CSP + 1
$a0,$0f	(CSP + 1) = (CSP + 1) − (CSP); CSP = CSP + 1
$a0,$10	(CSP + 1) = (CSP + 1) + (CSP); CSP = CSP + 1
$a0,$11	(CSP + 1) = (CSP + 1) .eq. (CSP); CSP = CSP + 1
$a0,$12	(CSP + 1) = (CSP + 1) .ne. (CSP); CSP = CSP + 1
$a0,$13	(CSP + 1) = (CSP + 1) .lt. (CSP); CSP = CSP + 1
$a0,$14	(CSP + 1) = (CSP + 1) .le. (CSP); CSP = CSP + 1
$a0,$15	(CSP + 1) = (CSP + 1) .gt. (CSP); CSP = CSP + 1
$a0,$16	(CSP + 1) = (CSP + 1) .ge. (CSP); CSP = CSP + 1
$a0,$17	(CSP + 1) = (CSP + 1) .slt. (CSP); CSP = CSP + 1
$a0,$18	(CSP + 1) = (CSP + 1) .sle. (CSP); CSP = CSP + 1
$a0,$19	(CSP + 1) = (CSP + 1) .sgt. (CSP); CSP = CSP + 1
$a0,$1a	(CSP + 1) = (CSP + 1) .sge. (CSP); CSP = CSP + 1
$a0,$1b	(CSP + 1) = (CSP + 1) .and. (CSP); CSP = CSP + 1
$a0,$1c	(CSP + 1) = (CSP + 1) .xor. (CSP); CSP = CSP + 1
$a0,$1d	(CSP + 1) = (CSP + 1) .or. (CSP); CSP = CSP + 1

◆ 制　御

オブジェクトコマンドの終了を示したり，セクションの変更等を行います。

コード `.dc.b $00,$00`

機　能 オブジェクトファイルの終わりを示します。アセンブラの `.end` 疑似命令[2]に相当します。

2)「プログラムの終了指定」を参照 (第3.5.1節 P.98)。

コード .dc.b $20,SECTION
.dc.l 0

機 能 セクションを変更します。アセンブラの .text 疑似命令[3]などに相当します。

3)「テキストセクションの宣言」を参照(第 3.5.2 節 P.101)。

コード .dc.b $e0,$01
.dc.w SECTION
.dc.l ADDRESS

機 能 オブジェクトファイルのリンクを要求します。アセンブラの .request 疑似命令[4]に相当します。

4)「リンク時のライブラリ指定」を参照(第 3.5.1 節 P.98)。

◆ **書き込み**

外部参照シンボルの値を書き込んだり，定数データを書き込みます。

コード .dc.b $10,SIZE-1
.dc.b (data),(data), ... ,(data)
.even

機 能 定数データを書き込みます。
データは 1 ～ 256 バイトまで書き込め，256 バイトを超えるものは分割します。アセンブラの .dc.b 疑似命令[5]などに相当します。

5)「定数データの定義」を参照(第 3.5.6 節 P.114)。

コード .dc.b $30,$00
.dc.l SIZE

機 能 メモリ領域を確保します。アセンブラの .ds.b 疑似命令[6]などに相当します。text，data， rdata，rldata セクションの場合は，確保した領域を $00 で埋めます。それ以外のセクションのときは，大きさを確保するようにします。

6)「メモリ領域の確保」を参照(第 3.5.6 節 P.115)。

コード .dc.b {$43, $47},XREF_LABEL_SECTION
.dc.w LABEL_NO

機 能 外部参照のラベルの値をバイトサイズの値とみなして書き込みます。
書き込む値が -$80 ～ $ff の範囲に入っていない場合は，Over flow エラーになります。書き込む値の属性がアドレスを示す場合は，Relative エラーになります。

コード　`.dc.b $43,SECTION`
　　　　`.dc.l ADDRESS`

機　能　SECTION:ADDRESS をバイトサイズの値とみなして書き込みます。
HLK では絶対アドレス形式の実行ファイルをサポートしていないので，Relative エラーになります。

コード　`.dc.b {$41, $45},XREF_LABEL_SECTION`
　　　　`.dc.w LABEL_NO`

機　能　外部参照のラベルの値をワードサイズの値とみなして書き込みます。
書き込む値が -$8000 ～ $ffff の範囲に入っていない場合は，Over flow エラーになります。書き込む値の属性がアドレスの場合は，Relative エラーになります。

コード　`.dc.b $41,SECTION`
　　　　`.dc.l ADDRESS`

機　能　SECTION:ADDRESS をワードサイズの値とみなして書き込みます。
HLK では絶対アドレス形式の実行ファイルをサポートしていないので，Relative エラーになります。

コード　`.dc.b {$42, $46},XREF_LABEL_SECTION`
　　　　`.dc.w LABEL_NO`

機　能　外部参照のラベルの値をロングワードサイズの値とみなして書き込みます。

コード　`.dc.b $42,SECTION`
　　　　`.dc.l ADDRESS`

機　能　SECTION:ADDRESS をロングワードサイズの値とみなして書き込みます。

コード　`.dc.b $40,XREF_LABEL_SECTION`
　　　　`.dc.w LABEL_NO`

機　能　外部参照のラベルの値をワードサイズの値とみなして書き込みます。
書き込む値が -$80 ～ $ff の範囲に入っていない場合は，Over flow エラーになります。書き込む値の属性がアドレスの場合は，Relative エラーになります。

コード .dc.b $40,SECTION
.dc.l ADDRESS

機　能 SECTION:ADDRESS をワードサイズの値とみなして書き込みます。
HLK では，絶対アドレス形式の実行ファイルをサポートしていないのでエラーになります。また，書き込む値が -$80 ～ $ff の範囲に入っていない場合は Over flow エラーになり，書き込む値の属性がアドレスの場合は Relative エラーになります。

コード .dc.b {$53, $57},XREF_LABEL_SECTION
.dc.w LABEL_NO
.dc.l OFFSET

機　能 $43, $47 とほとんど同じ機能ですが，OFFSET 値が加算されます。OFFSET が 0 の場合は $43, $47 と同じ機能になります。

コード .dc.b $53,SECTION
.dc.l ADDRESS

機　能 $43 とほとんど同じ機能ですが，OFFSET 値が加算されます。OFFSET が 0 の場合は $43 と同じ機能になります。

コード .dc.b {$51, $55},XREF_LABEL_SECTION
.dc.w LABEL_NO
.dc.l OFFSET

機　能 $41, $45 とほとんど同じ機能ですが，OFFSET 値が加算されます。OFFSET が 0 の場合は $41, $45 と同じ機能になります。

コード .dc.b $51,SECTION
.dc.l ADDRESS

機　能 $41 とほとんど同じ機能ですが，OFFSET 値が加算されます。OFFSET が 0 の場合は $41 と同じ機能になります。

コード .dc.b {$52, $56},XREF_LABEL_SECTION
.dc.w LABEL_NO
.dc.l OFFSET

機　能 $42，$46 とほとんど同じ機能ですが，OFFSET 値が加算されます。OFFSET が 0 の場合は $42，$46 と同じ機能になります。

コード .dc.b {$52, $56},SECTION
.dc.l ADDRESS

機　能 $42，$46 とほとんど同じ機能ですが，OFFSET 値が加算されます。OFFSET が 0 の場合は $42，$46 と同じ機能になります。

コード .dc.b {$65, $69},SECTION
.dc.l ADDRESS
.dc.w LABEL_NO

機　能 外部参照のラベルの値から，SECTION:ADDRESS の値を引いた値をワードサイズの値とみなして書き込みます。
書き込む値が -$8000 ～ $7fff の範囲に入っていない場合は，Over flow エラーになります。書き込む値の属性がアドレスの場合は，Relative エラーになります。

6.1.4　ライブラリファイルのフォーマット

ライブラリファイルには，アーカイブ形式とライブラリアン形式 がありますが，ここではアーカイブ形式のフォーマットについて説明します。ライブラリアン形式については，一部解析されていない部分がありますので割愛します。

アーカイブ形式のフォーマットは単純で，アーカイブヘッダとオブジェクトファイルにヘッダをつけたものの集合になっています。これを，図で表すと Fig. 6-2 のようになります。

そして アーカイブヘッダは Fig. 6-3，オブジェクトヘッダは Fig. 6-4 のようになっています。そしてアーカイブヘッダは Fig. 6-3，オブジェクトヘッダは Fig. 6-4 のようになっています。また，Fig. 6-4 中の略号の意味は Table 6-4 のとおりです。

アーカイブヘッダ
オブジェクトヘッダ
オブジェクトファイル
⋮ ⋮
オブジェクトヘッダ
オブジェクトファイル

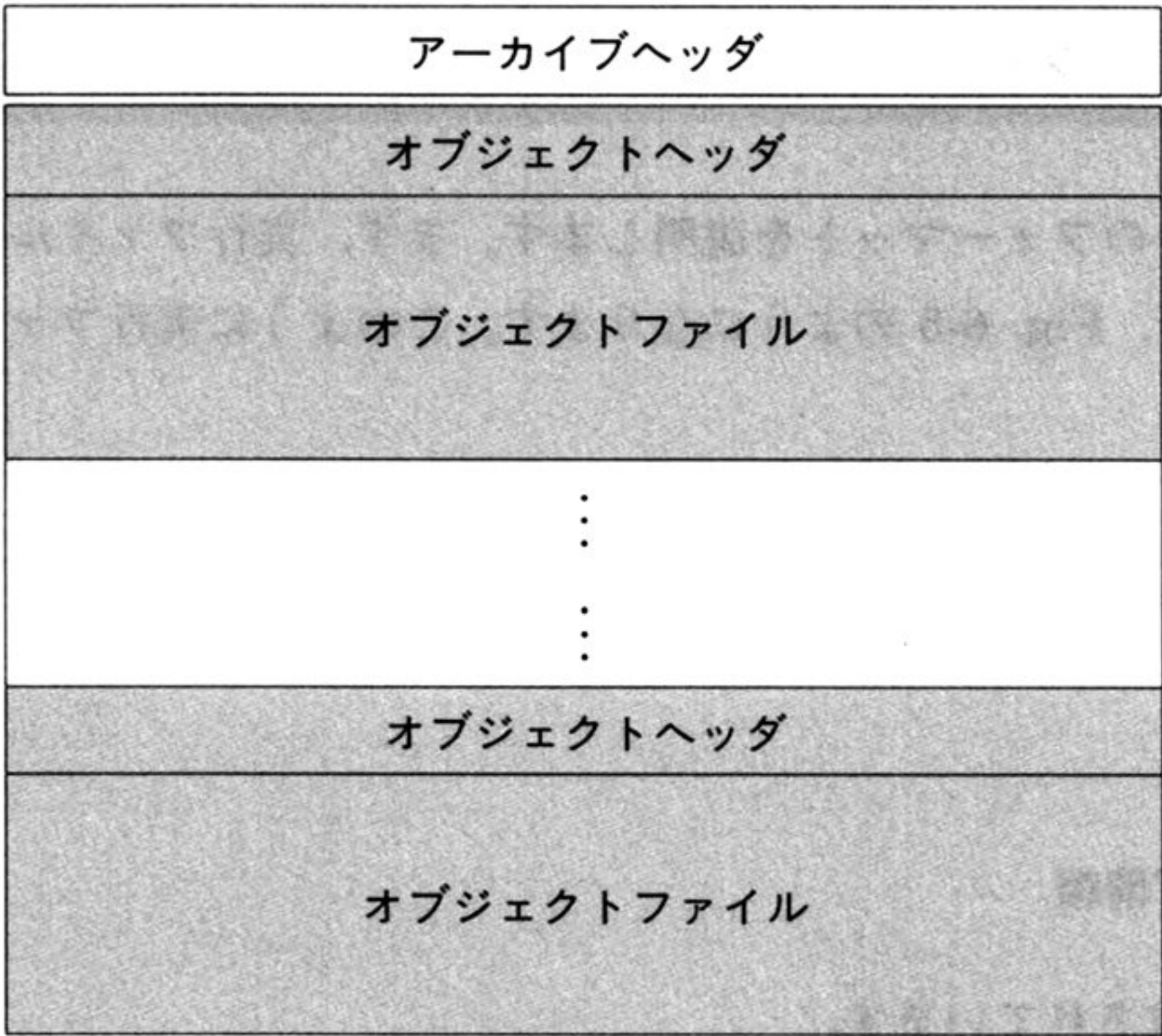



Fig. 6-2● アーカイブ形式のファイルフォーマット

+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0a	+0b	+0c	+0d	+0e	+0f
d1	00	00	00	00	02										

Fig. 6-3● アーカイブヘッダのフォーマット

+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0a	+0b	+0c	+0d	+0e	+0f
object name															
							00	size				date			

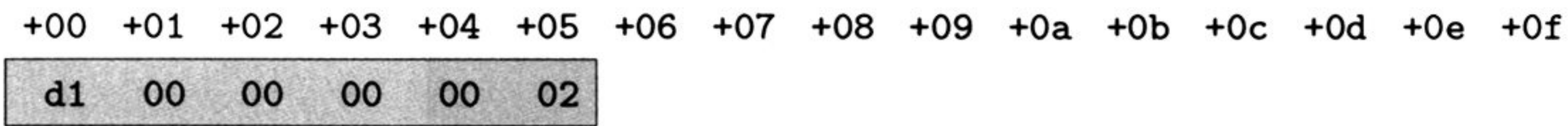
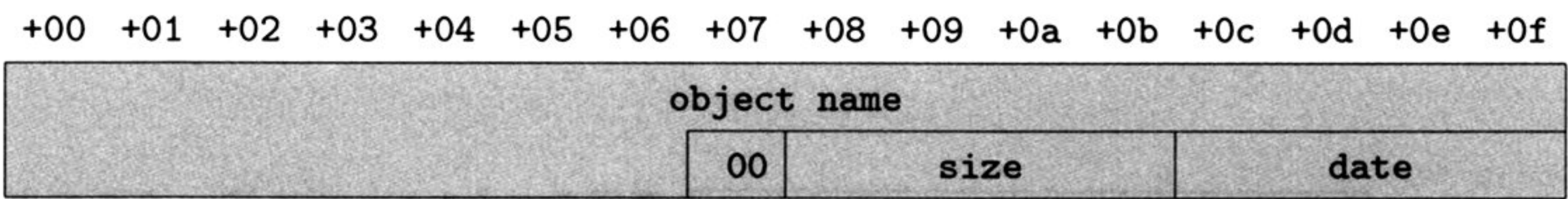



Fig. 6-4● オブジェクトヘッダのフォーマット

Table 6-4● Fig. 6-4 中の略号の意味

略 号	意 味
object name	オブジェクトファイルの名前。23 文字未満のときは残りを 0 で埋める
size	オブジェクトファイルのサイズ。単位はバイト
date	オブジェクトファイルの日付。フォーマットは DOS コールの FILEDATE($FF57) が返すフォーマットと同じ

6.1.5 実行ファイルのフォーマット

ここでは，実行ファイルのフォーマットを説明します。まず，実行ファイルのフォーマットを図に示すと，Fig. 6-5 のようになります。このように実行ファイルは，

- 実行ファイルヘッダ
- プログラム本体
- リロケートテーブル
- シンボル情報
- シンボリックデバッグ情報

という 5 つの部分から構成されています。

実行ファイルヘッダ
プログラム本体
リロケートテーブル
シンボル情報
シンボリックデバッグ情報

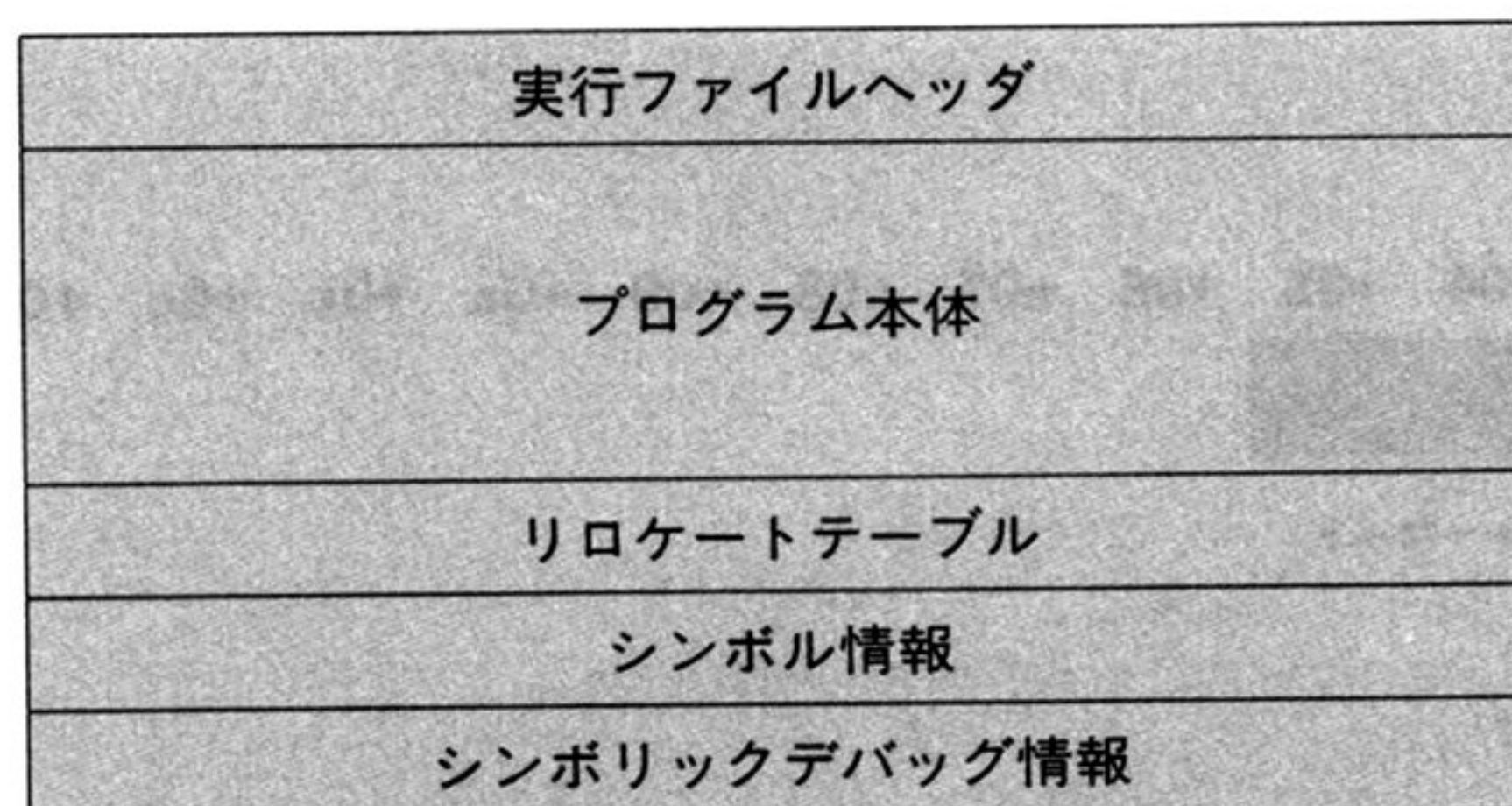



Fig. 6-5● 実行ファイルのフォーマット

◆ 実行ファイルヘッダ

実行ファイルヘッダは Fig. 6-6 のようになっています。実行するときは，このヘッダの先頭を見て実行ファイルであることを確認します。また Fig. 6-6 中の略号の意味は，Table 6-5 のとおりです。

+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0a	+0b	+0c	+0d	+0e	+0f
48	55	00	00	00	00	00	00	exec address				text size			
data size				bss size				rel. table size				symbol info. size			
SCD line size				SCD symbol size				SCD string size							
reserved															

Fig. 6-6● 実行ファイルのヘッダ

Table 6-5● Fig. 6-6 中の略号の意味

略　号	意　味
exec address	実行アドレス
text size	テキストセクションのサイズ
data size	データセクションのサイズ
bss size	bss, common, stack セクションのサイズの合計
rel. table size	リロケートテーブルのサイズ
symbol info. size	シンボルテーブルのサイズ
SCD line size	シンボリックデバッグ情報の行番号テーブルのサイズ
SCD symbol size	シンボリックデバッグ情報のシンボルテーブルのサイズ
SCD string size	シンボリックデバッグ情報の文字列テーブルのサイズ

◆ リロケートテーブル

実行ファイルが実行されるときは，メモリのどこかにプログラムがロードされています。ロードされる位置はメモリの使用状況によって異なり，固定ではありません。しかしほとんどのプログラムは，どこか決まった位置にロードされたときにしか実行できないようになっています[7]。この問題を解決するのが，リロケートテーブルです。このリロケートテーブルを使って，実行ファイル内のロードされる位置に依存する部分を補正します。

7)ロードされる位置にかかわらず実行できるプログラムもあります。それは，拡張子が '.r' のファイルです。このようなコードは，リロケータブルな実行ファイルとか，PIC(Position Independent Code と呼ばれることもあります。

　リロケートテーブルには，前回補正した位置からの差が入っています。テーブルの値と実際の差の値の関係を Table 6-6 に示します。一番最初に補正した場所をプログラムの先頭とします。

Table 6-6● テーブルの値と実際の差の値

テーブルの値		実際の差の値
x	(x は 2 バイトの偶数値)	$00000000 $\leq x \leq$ $0000fffe
$0001 x	(x は 4 バイトの偶数値)	$00010000 $\leq x \leq$ $fffffffe

◆ シンボル情報

シンボル情報は，実行ファイルを実行するときには使用されません。この情報はプログラムをデバッグするときに，デバッガが使用する情報です。シンボル情報には「シンボルの名前」,「値」,「属性」[8]が入っています。これらのフォーマットを Fig. 6-7 に示します。また図中の type はシンボルの属性を表す値で，Table 6-7 のようになっています。

8)「属性」を参照(第 4.5.1 節 P.155)。

+00	+01 +02 +03 +04	+05 +06 +07 ··· +??	+??	+??
type	value	symbol name	00	even
⋮				
type	value	symbol name	00	even

Fig. 6-7● シンボル情報のフォーマット

Table 6-7 ● 値と属性の関係

値 (16 進数)	属　性
0003	common　rcommon　rlcommon
0200	absolute　rdata　rbss　rstack　rldata　rlbss　rlstack
0201	text
0202	data
0203	bss
0204	stack

◆ シンボリックデバッグ情報

シンボリックデバッグ情報もまた，シンボル情報のようにデバッガが使用する情報です。しかしシンボル情報と異なり，シンボリックデバッグ情報には元のソースプログラム (現在は，C 言語だけ) の情報が入っています。詳しくは，次節を参照してください。

◆ 相対オフセットテーブル

相対オフセットテーブルは，相対セクションを初期化するときに，アドレスに依存する部分を補正するためのテーブルです。たとえば SX–Window で，次のような C のプログラムを実行することを考えてみましょう。

List 6-1 ● sample.c

```
1:  remote int table[20];
2:  remote int *table_ptr = table;
3:
4:  void main (int argc, char **argv)
5:  {
6:    /* SX-Window program */
7:  }
```

List 6-1 の場合，`table_ptr` の値は `table` のアドレスに依存することがわかります。`table` の位置は，このプログラムが相対セクションの領域を確保するまでわかりません。そこで，`table_ptr` の値を補正する必要があるわけです。

相対オフセットテーブルのフォーマットには，補正しなければならない相対セクションのアドレスが書いてあるだけです。しかし，相対オフセットテーブルには自分自身の大きさの情報は含まれていません。この情報は，セクション情報内に書かれています。

実行ファイル内での相対セクションの位置を，Fig. 6-8 に示しておきます。

実行ファイルヘッダ
text セクションの内容
セクション情報
rdata セクションの内容
rldata セクションの内容
相対オフセットテーブル
data セクションの内容
⋮ ⋮ ⋮

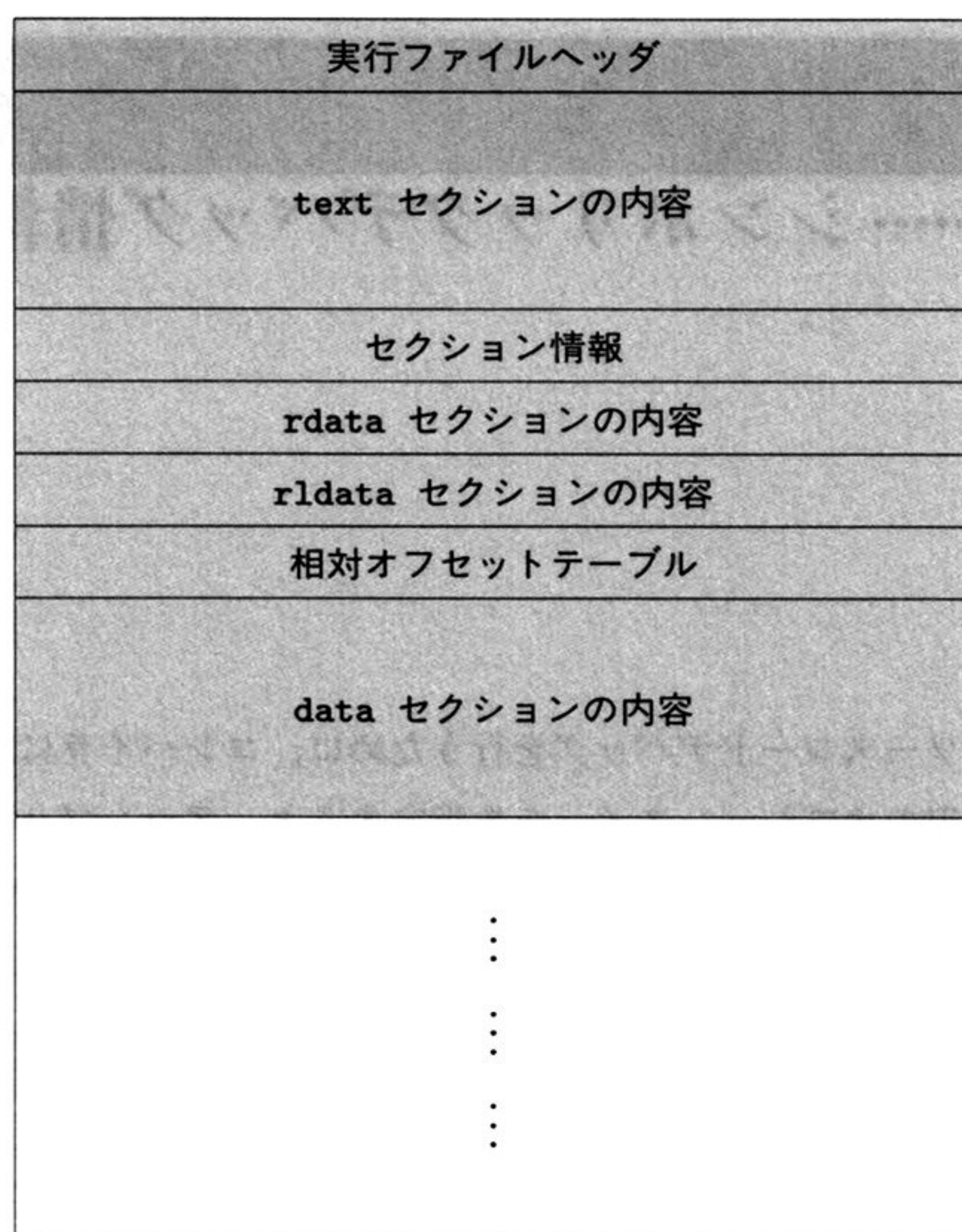



Fig. 6-8● 実行ファイル内での相対セクションの位置

6.2……………………シンボリックデバッグ情報

C言語プログラムなどのソースコードデバッグを行うために，コンパイラに対してソースコードデバッグ用のオプションスイッチを指定すると，アセンブリ言語にシンボリックデバッグ用の疑似命令が出力されます。そして，それによってオブジェクトファイルや実行ファイルにシンボリックデバッグ情報が出力されるようになります。このセクションでは，これらのファイル上でのシンボリックデバッグ情報のフォーマットについて説明します[1]。

6.2.1 シンボリックデバッグ情報の構成

シンボリックデバッグ情報は，オブジェクトファイルや実行ファイル中では次の3つの部分に分かれています[2]。

- 行番号テーブル
  .ln 疑似命令[3]で指定した行番号とロケーションとの対応を格納します。
- シンボルテーブル
  .def ～ .endef 疑似命令[4]によって作成した，シンボルテーブルエントリの内容を格納します。
- 文字列テーブル
  シンボルテーブルの中では，シンボル名を格納する領域は8バイトに制限されています。そこで長さが8バイトを超えるシンボル名の場合は，この文字列テーブルにシンボル名を格納します。

これらの情報の格納方法は，オブジェクトファイルと実行ファイルとでは少々異なっています。次にそれぞれの格納方法について説明します。

◆ オブジェクトファイルの場合

シンボリックデバッグ情報をもたないオブジェクトファイルは，終了コマンド“00 00”で終了します。シンボリックデバッグ情報は，この終了コマンドの次にFig. 6-9のように格納されます(P.238 Fig. 6-1も参照)。

1)シンボリックデバッグ情報については SHARP から公開されている資料が存在しないため，このセクションの内容，および HAS への実装は，筆者(中村)が AS.X Ver 2.0 の出力を独自に解析した結果がもとになっています。

2)余談ですが，この構成および各テーブルのフォーマットは，UNIX System V の共通オブジェクトフォーマット(COFF フォーマット)がもとになっているようです。はじめからこのことを知っていれば，解析もずいぶん楽だったのですが・・・

3)「行番号とロケーションの対応の出力指定」を参照(第3.5.9節 P.124)。

4)「シンボルテーブルエントリの作成」を参照(第3.5.9節 P.125)。

オブジェクトファイルの情報
各セクションの情報
(指定されていれば) プログラムの実行アドレス, オブジェクトファイルのリンク要求
シンボルの定義
上記以外のコマンド
終了コマンド
シンボリックデバッグ情報

SCD line size
SCD symbol size
SCD string size
行番号テーブル ...
シンボルテーブル ...
文字列テーブル ...

Fig. 6-9● オブジェクトファイル中のシンボリックデバッグ情報

Table 6-8● Fig. 6-9の略号の意味

略号	意味
SCD line size	シンボリックデバッグ情報の行番号テーブルのサイズ
SCD symbol size	シンボリックデバッグ情報のシンボルテーブルのサイズ
SCD string size	シンボリックデバッグ情報の文字列テーブルのサイズ

◆ 実行ファイルの場合

実行ファイルで，シンボリックデバッグ情報をもつことができるのは.X形式[5]のファイルにかぎられています。

シンボリックデバッグ情報は，その長さがFig. 6-6 (P.250) のように実行ファイルのヘッダに格納され，デバッグ情報本体は，Fig. 6-10のように単に .X形式実行ファイルの最後につながった形で格納されています[6](P.250 Fig. 6-5も参照)。

5)再配置情報をもつ，ソフトウェアリロケータブル形式の実行ファイルです。実行ファイルには，その他に .R 形式 (フルリロケータブル形式)，.Z 形式 (固定アドレス形式) がありますが，これらのファイルはシンボリックデバッグ情報をもつことができません。

6)シンボリックデバッグ情報中のアドレスデータは，リンカによって結合された後のアドレスに変更されています。

実行ファイルヘッダ
プログラム本体
リロケートテーブル
シンボル情報
シンボリックデバッグ情報

行番号テーブル ...
シンボルテーブル ...
文字列テーブル ...

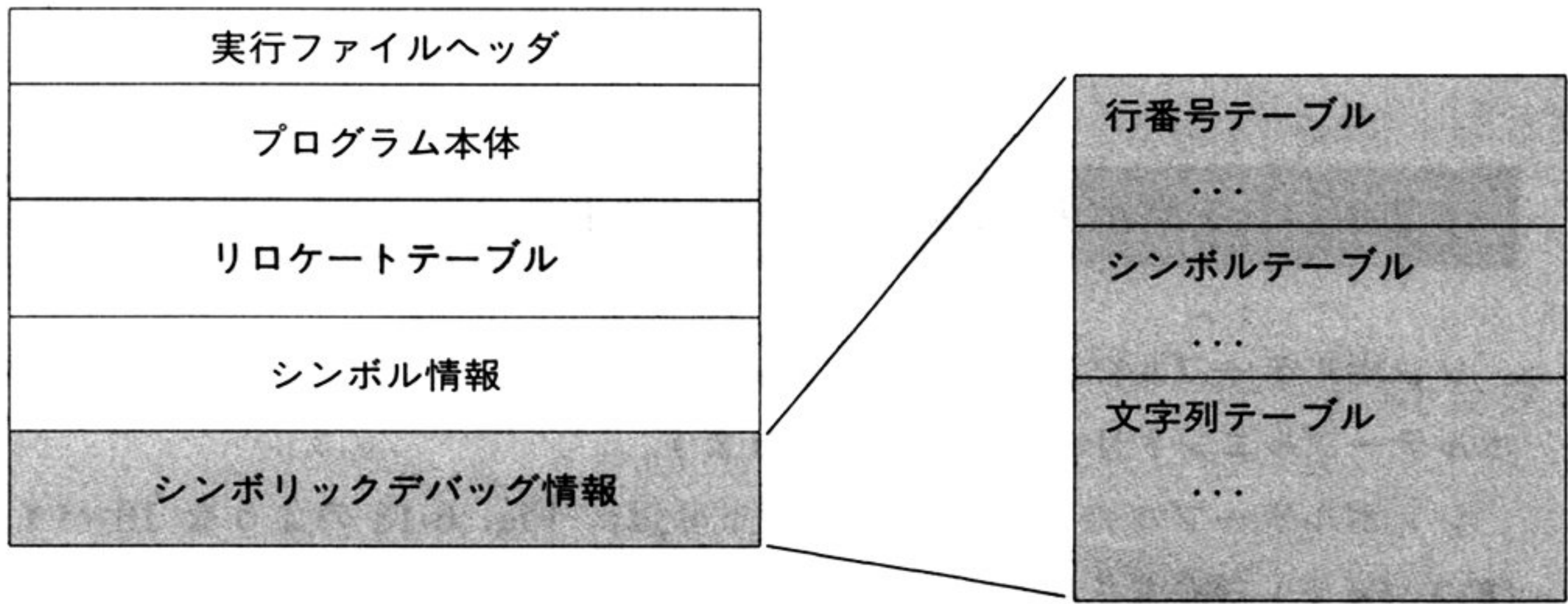



Fig. 6-10● 実行ファイル中のシンボリックデバッグ情報

6.2.2 行番号テーブルのフォーマット

7)「行番号とロケーションの対応の出力指定」を参照(第3.5.9節 P.124)。

行番号テーブルには，.ln 疑似命令[7]によって指定した行番号とロケーションとの対応が格納されます。行番号テーブルの1つのエントリは，Fig. 6-11のような6バイトで構成されています。

オフセット	内容
+$00.l	*location* (ロケーション値)
+$04.w	*line* (行番号)

Fig. 6-11● 行番号テーブルの1エントリ

8)行番号は，C言語ソースプログラムの関数定義が始まる行を1行目として数えます。

9)機械語命令の存在するアドレスと考えてください。

10)シンボルテーブルの各エントリの番号です。

ここで *line* は .ln 疑似命令によって指定した行番号[8]を，そして *location* はその行番号に対応するロケーション値[9]を表します。

また，各関数定義が始まる行では，*line* が0であるエントリが作成されます。この場合，対応する *location* の内容は，シンボルテーブル上のその関数名エントリのインデックス番号[10]になります。

したがって行番号テーブル全体としては，Fig. 6-12のような構成になるわけです。

location	*line*
シンボルテーブルのインデックス番号	0
ロケーション値	行番号
ロケーション値	行番号
...	...
シンボルテーブルのインデックス番号	0
ロケーション値	行番号
ロケーション値	行番号
...	...

関数1つに対応

Fig. 6-12● 行番号テーブル全体の構成

6.2.3 シンボルテーブルのフォーマット

11)「シンボルテーブルエントリの作成」を参照(第3.5.9節 P.125)。

シンボルテーブルには，.def ～ .endef 疑似命令[11]によって作成されたシンボルテーブルエントリの内容が格納されます。

シンボルテーブルの1つの基本エントリは，Fig. 6-13のような18バイト($12バイト)で構成されています。エントリの各フィールドの意味は次のとおりです。

オフセット	内　容	
+$00 ⋮ +$07	*name*	(シンボル名)
+$08.l	*value*	(シンボルのもつ値)
+$0C.w	*sect*	(値の存在するセクション)
+$0E.w	*type*	(シンボルの型)
+$10.b	*scl*	(シンボルの記憶クラス)
+$11.b	*auxent*	(補助エントリの有無)

Fig. 6-13● シンボルテーブルの基本エントリ

- *name* (シンボル名)
  シンボルの名前を格納します。シンボル名の長さが 8 バイト以下の場合は，ここに直接 ASCII コードで格納され，余りは $00 で埋められます。
  シンボルの名前が 8 バイトを超える場合は，実際のシンボル名は文字列テーブルに格納されます。このとき最初の 4 バイトは $00 で埋められ，残りの 4 バイトには，実際のシンボル名を格納している文字列テーブルの先頭アドレスからの OFFSET 値がロングワードサイズで格納されます。
- *value* (シンボルのもつ値)
  このシンボルのもつ値を格納します。.val 疑似命令[12]で指定された値そのものです。
- *sect* (値の存在するセクション)
  value の存在するセクションを格納します。*sect* の値とセクションとの対応は，Table 6-9 のとおりです。

12)「シンボルの値の指定」を参照 (第 3.5.9 節 P.126)。

Table 6-9● sect の値とセクションとの対応

値	セクション
$0001	.text *value* はテキストセクションの値
$0002	.data *value* はデータセクションの値
$0003	.bss *value* はブロックストレージセクションの値
$00FE	(コモンラベル) シンボルはコモンラベルであり，*value* はコモンラベルの番号
$FFFE	(列挙メンバ名) シンボルは列挙メンバ名であり，*value* はそのメンバのもつ値
$FFFF	(auto 変数/構造体メンバ名) *value* は auto 変数のスタック上の OFFSET 値，または構造体の先頭アドレスからの OFFSET 値

- *type* (シンボルの型)
  シンボルの C 言語における型を格納します。`.type` 疑似命令[13]によって指定された値そのものです。
- *scl* (シンボルの記憶クラス)
  シンボルの記憶クラスを格納します。`.scl` 疑似命令[14]によって指定された値そのものです。
- *auxent* (補助エントリの有無)
  このエントリの後ろに補助エントリがある場合は $01，ない場合は $00 が入ります。

13)「C 言語における型の宣言」を参照 (第 3.5.9 節 P.127)。

14)「記憶クラスの宣言」を参照 (第 3.5.9 節 P.126)。

各エントリには $0, 1, 2, \cdots$ と順番にインデックス番号がつけられ，あるエントリが他のエントリを指すとき[15]にはこの番号が使われます。

シンボルの種類によっては，その情報を格納するのに 1 つのエントリのみでは不足する場合があります。このときは，その次のエントリを補助エントリとして使用します。補助エントリの内容はシンボルの種類によってまちまちなので，以下にシンボルの種類別にその内容を説明します。

15)たとえばある構造体変数に対応するエントリが，その構造を表すタグ名に対応するエントリを指す場合などです。

◆ 変　数

変数が構造をもたない型[16]であるときは，補助エントリは不要です。補助エントリは，変数が構造体や配列である場合に使用されます。

補助エントリを含めたエントリ全体の内容は，Fig. 6-14 のとおりです。各フィールドの意味は次のようになります。

16)“スカラ型” といいます。int や char，そしてすべてのポインタ変数などがこれにあたります。

- *tag* (タグ名へのインデックス)
  シンボルが構造体や共用体，列挙型であれば場合，その構造はタグ名のエントリ[17]から格納されています。これはそのエントリのインデックス番号を格納します。
- *size* (シンボルのサイズ)
  シンボルのサイズを格納します。`.size` 疑似命令[18]によって指定された値そのものです。
- *dim1* ～ *dim4* (配列の要素数)
  シンボルが配列であった場合にその要素数を格納します。`.dim` 疑似命令[19]によって指定された値そのものです。

17).tag 疑似命令によって指定されたシンボルのエントリになります。

18)「サイズの指定」を参照 (第 3.5.9 節 P.129)。

19)「配列の指定」を参照 (第 3.5.9 節 P.130)。

残りのフィールドは未使用です。未使用のフィールドは，原則として 0 にしておきます。

オフセット	内　容
+$00 ⋮ +$07	*name*
+$08.l	*value*
+$0C.w	*sect*
+$0E.w	*type*
+$10.b	*scl*
+$11.b	*auxent*　（必ず $01 が入る）
+$12.l	*tag*　（タグ名へのインデックス）
+$16.l	*size*　（シンボルのサイズ）
+$1A.w	*dim1*　（配列の要素数）
+$1C.w	*dim2*　（配列の要素数）
+$1E.w	*dim3*　（配列の要素数）
+$20.w	*dim4*　（配列の要素数）
+$22.w	（未使用）

Fig. 6-14● 構造体/配列のエントリ

◆ タ　グ

構造体や共用体の構造を表すためにはタグ名を使用しますが[20]，シンボルテーブルでの構造の宣言は，このタグ名のエントリから開始します。

タグ名のエントリの後ろには，その構造のメンバ名のエントリが並び[21]，最後に特殊シンボル“.eos”[22]のエントリによって構造の宣言を終了します。

これらのエントリの関係は，Fig. 6-15 のようになります。以下に，各エントリの内容を説明します。

20)C 言語では，タグ名のない構造体や共用体の定義も可能です。この場合，コンパイラはその構造に対して仮の名前を与えます。

21)各メンバは変数なので，すでに説明した変数のエントリと同じものが並ぶことになります。

22)“構造の終了”（end of structure）の略と思われます。

ソースプログラム

```
  ⋮
struct foo {
    int x;
    int y;
      ⋮
}
  ⋮
```

→

シンボルテーブル

⋮
- タグ名 foo のエントリ
- int x のエントリ
- int y のエントリ

⋮
- .eos のエントリ

⋮

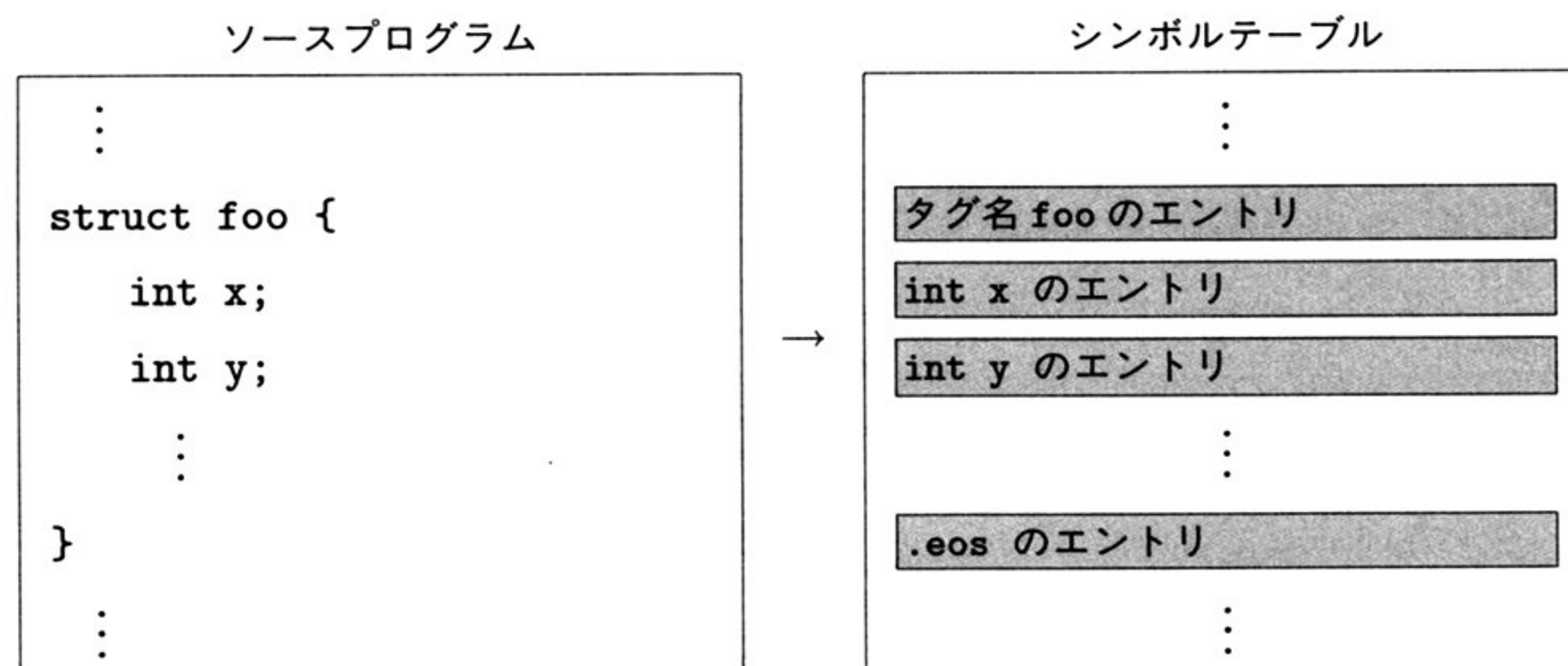



Fig. 6-15● 構造体とシンボルエントリとの関係

❖ タグ名のエントリ

タグ名のエントリは，必ず補助エントリを使用します (Fig. 6-16)。各フィールドの意味は次のとおりです。

- *size* (構造のサイズ)
  このタグ名の示す構造のサイズを格納します。`.size` 疑似命令[23]によって指定された値そのものです。
- *next* (次のタグ名へのインデックス)
  シンボルテーブル上で，このタグ名の次にあるタグ名のエントリのインデックス番号を格納します。

23)「サイズの指定」を参照（第3.5.9節 P.129）。

オフセット	内　容	
+$00 : +$07	*name*	
+$08.l	(未使用)	
+$0C.w	*sect*	(必ず $FFFE が入る)
+$0E.w	*type*	
+$10.b	*scl*	
+$11.b	*auxent*	(必ず $01 が入る)
+$12.l	(未使用)	
+$16.l	*size*	(構造のサイズ)
+$1A.l	(未使用)	
+$1E.l	*next*	(次のタグ名へのインデックス)
+$22.w	(未使用)	

Fig. 6-16● タグ名のエントリ

❖ .eos のエントリ

.eos シンボルのエントリは，必ず補助エントリを使用します (Fig. 6-17)。各フィールドの意味は次のとおりです。

- *tag* (対応するタグ名へのインデックス)
  シンボルテーブル上で，この .eos に対応するタグ名のエントリのインデックス番号を格納します。
- *size* (構造のサイズ)
  構造のサイズを格納します。.size 疑似命令によって指定された値そのものです。

オフセット	内　容	
+$00 ⋮ +$07	*name*	(.eos に固定)
+$08.l	*value*	
+$0C.w	*sect*	(必ず $FFFF が入る)
+$0E.w	(未使用)	
+$10.b	*scl*	(必ず $66 が入る)
+$11.b	*auxent*	(必ず $01 が入る)
+$12.l	*tag*	(対応するタグ名へのインデックス)
+$16.l	*size*	(構造のサイズ)
+$17 ⋮ +$23	(未使用)	

Fig. 6-17● .eos シンボルのエントリ

◆ 関　数

シンボルテーブルでの関数定義は，関数名のエントリから開始します。関数名のエントリに続いて特殊シンボル“.bf”[24]のエントリがあり，その後ろに関数の引数や関数内のローカル変数のエントリが並びます[25]。そして，最後に特殊シンボル“.ef”[26]のエントリによって関数定義が終了します。

関数の中にはさらに関数内ブロックがあり，そのブロック内でローカル変数を定義することもできます。この場合は，そのブロック構造に応じて特殊シンボル“.bb”[27]と“.eb”[28]のエントリがおかれ，ブロック内ローカル変数のエントリは，これらのエントリの間に入ります。

これらのエントリの関係は，Fig. 6-18 のようになります。以下に，各エントリの内容を説明します。

24)“関数の開始”(beginning of function)の略と思われます。

25)これらのエントリの内容は，すでに説明した変数のエントリと同じです。

26)“関数の終了”(end of function)の略でしょう。

27)“ブロックの開始”(beginning of block)の略でしょう。

28)“ブロックの終了”(end of block)の略でしょう。

ソースプログラム

```
  :
void foo(int x)
{
    int i;
      :
    {
        int j;
          :
    }
      :
}
  :
```

→

シンボルテーブル

:
void foo() のエントリ
.bf のエントリ
int x のエントリ
int i のエントリ
:
.bb のエントリ
int j のエントリ
:
.eb のエントリ
:
.ef のエントリ
:

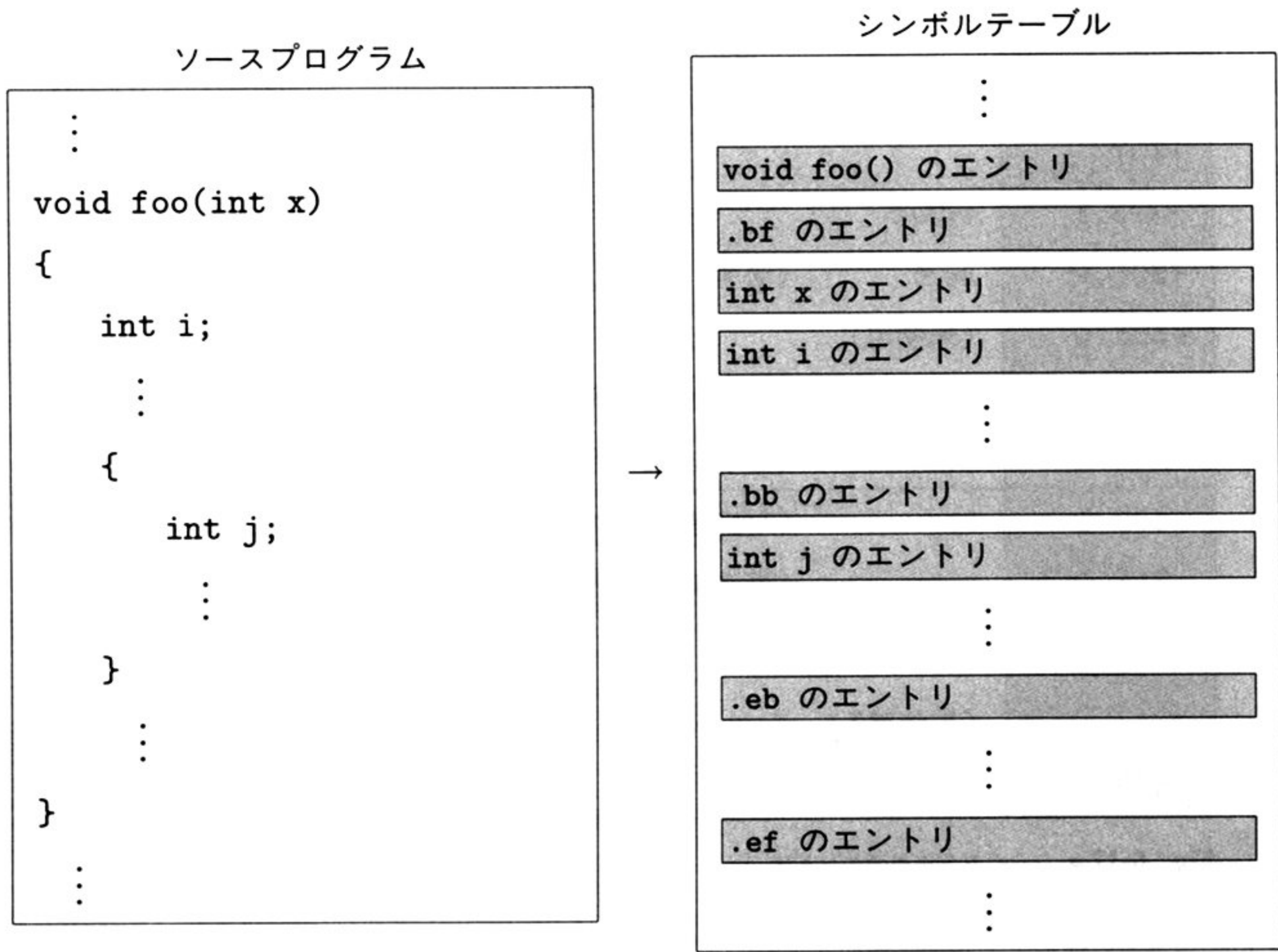



Fig. 6-18● 関数構造とシンボルエントリとの関係

❖ 関数名のエントリ

関数名のエントリは，必ず補助エントリを使用します (Fig. 6-19)。各フィールドの意味は次のとおりです。

- *size* (関数のサイズ)
  この関数のサイズ[29]を格納します。
- *lpos* (行番号テーブルのオフセット)
  関数定義の始まる行では，行番号テーブルに *line* が 0 であるエントリが作成されます[30]。*lpos* には，行番号テーブル先頭からこのエントリへの OFFSET 値が格納されます。
- *next* (次の関数名へのインデックス)
  シンボルテーブル上で，この関数名の次にある関数名のエントリのインデックス番号を格納します。

29)関数の機械語命令そのものの大きさです。

30)第 6.2.2 節 (P.256) 参照。

オフセット	内　容	
+$00 : +$07	*name*	
+$08.l	*value*	
+$0C.w	*sect*	(必ず $0001 が入る)
+$0E.w	*type*	
+$10.b	*scl*	
+$11.b	*auxent*	(必ず $01 が入る)
+$12.l	(未使用)	
+$16.l	*size*	(関数のサイズ)
+$1A.l	*lpos*	(行番号テーブルのオフセット)
+$1E.l	*next*	(次の関数名へのインデックス)
+$22.w	(未使用)	

Fig. 6-19 ● 関数名のエントリ

❖ .bf のエントリ

.bf シンボルのエントリは，必ず補助エントリを使用します(Fig. 6-20)。各フィールドの意味は次のとおりです。

- *line* (関数の行番号)
  C言語のソースプログラムで，この関数定義が始まっている行の行番号[31]を格納します。.line 疑似命令[32]によって指定された値そのものです。
- *efpos* (対応する .ef へのインデックス)
  シンボルテーブル上で，この .bf に対応する .ef のエントリのインデックス番号を格納します。

31)最初の行を1行目として数えます。

32)「行番号の指定」を参照(第3.5.9節 P.129)。

オフセット	内容	
+$00 ⋮ +$07	*name*	(.bf に固定)
+$08.l	*value*	
+$0C.w	*sect*	(必ず $0001 が入る)
+$0E.w	(未使用)	
+$10.b	*scl*	(必ず $65 が入る)
+$11.b	*auxent*	(必ず $01 が入る)
+$12.l	(未使用)	
+$16.w	*line*	(関数の行番号)
+$18.w	(未使用)	
+$1A.l	(未使用)	
+$1E.l	*efpos*	(対応する .ef へのインデックス)
+$22.w	(未使用)	

Fig. 6-20● .bf シンボルのエントリ

❖ .bb のエントリ

.bb シンボルのエントリは，必ず補助エントリを使用します (Fig. 6-21)。各フィールドの意味は次のとおりです。

- *line* (ブロックの行番号)
  C 言語ソースプログラムで，この関数内ブロックが開始する行の行番号[33]を格納します。.line 疑似命令[34]によって指定された値そのものです。
- *ebpos* (対応する .eb へのインデックス)
  シンボルテーブル上で，この .bb に対応する .eb の次の[35]エントリのインデックス番号を格納します。

33)関数定義の始まる行を 1 行目として数えます。.bf と行番号の意味が異なるので注意が必要です。

34)「行番号の指定」を参照 (第 3.5.9 節 P.129)。

35)なぜ“次の”エントリなのか，理由はわかりませんが ···。

オフセット	内　容	
+$00 : +$07	*name*	(.bb に固定)
+$08.l	*value*	
+$0C.w	*sect*	(必ず $0001 が入る)
+$0E.w	(未使用)	
+$10.b	*scl*	(必ず $64 が入る)
+$11.b	*auxent*	(必ず $01 が入る)
+$12.l	(未使用)	
+$16.w	*line*	(ブロックの行番号)
+$18.w	(未使用)	
+$1A.l	(未使用)	
+$1E.l	*ebpos*	(対応する .eb へのインデックス)
+$22.w	(未使用)	

Fig. 6-21 ● .bb シンボルのエントリ

❖ .eb のエントリ

.eb シンボルのエントリは，必ず補助エントリを使用します (Fig. 6-22)。フィールドの意味は次のとおりです。

- *line* (ブロックの行番号)
  C 言語ソースプログラムで，この関数内ブロックが終了する行の行番号を格納します。.line 疑似命令[36]によって指定された値そのものです。

36)「行番号の指定」を参照（第3.5.9節 P.129）。

オフセット	内　容	
+$00 ⋮ +$07	*name*	(.eb に固定)
+$08.l	*value*	
+$0C.w	*sect*	(必ず $0001 が入る)
+$0E.w	(未使用)	
+$10.b	*scl*	(必ず $64 が入る)
+$11.b	*auxent*	(必ず $01 が入る)
+$12.l	(未使用)	
+$16.w	*line*	(ブロックの行番号)
+$18 ⋮ +$23	(未使用)	

Fig. 6-22 ● .eb シンボルのエントリ

❖ .ef のエントリ

.ef シンボルのエントリは，必ず補助エントリを使用します (Fig. 6-23)。フィールドの意味は次のとおりです。

- *line* (関数の行数)
  C 言語ソースプログラム上での，この関数本体の行数を格納します。
  .line 疑似命令[37]によって指定された値そのものです。

37)「行番号の指定」を参照（第 3.5.9 節 P.129）。

オフセット	内　容	
+$00 〜 +$07	*name*	(.ef に固定)
+$08.l	*value*	
+$0C.w	*sect*	(必ず $0001 が入る)
+$0E.w	(未使用)	
+$10.b	*scl*	(必ず $65 が入る)
+$11.b	*auxent*	(必ず $01 が入る)
+$12.l	(未使用)	
+$16.w	*line*	(関数の行数)
+$18 〜 +$23	(未使用)	

Fig. 6-23● .ef シンボルのエントリ

◆ その他の特殊シンボル

これまでに説明してきた各種シンボルのほかに，いくつかの特殊シンボルがあります。ここでは，これら特殊シンボルのエントリの内容を説明します。

❖ .file のエントリ

.file 疑似命令[38]によって C 言語ソースプログラムのファイル名を指定すると，このエントリが作成されます。シンボルテーブルは必ずこのエントリから始まります[39]。

.file シンボルのエントリは，必ず補助エントリを使用します (Fig. 6-24)。各フィールドの意味は次のとおりです。

- *endpos* (エントリの終了位置)
  シンボルテーブルの最後にある，.bss シンボルの次のエントリ[40]のインデックス番号を格納します。
- *filename* (ソースファイル名)
  C 言語ソースプログラムのファイル名を格納します。領域は 14 バイトあり，14 バイトに満たないファイル名の場合は，余りが “$00” で埋められます。

38)「ソースファイル名の出力指定」を参照 (第 3.5.9 節 P.124)。

39) オブジェクトファイルでは .file エントリは 1 つのみですが，複数のオブジェクトファイルをリンクした実行ファイルの場合，リンクしたオブジェクトファイルの数だけ .file エントリが存在することになります。

40) 複数のオブジェクトファイルをリンクした実行ファイルの場合，これは次の .file シンボルのエントリになります。

オフセット	内容
+$00 ⋮ +$07	*name* (.file に固定)
+$08.l	*endpos* (エントリの終了位置)
+$0C.w	*sect* (必ず $FFFE が入る)
+$0E.w	(未使用)
+$10.b	*scl* (必ず $67 が入る)
+$11.b	*auxent* (必ず $01 が入る)
+$12 ⋮ +$1F	*filename* (ソースファイル名)
+$20 ⋮ +$23	(未使用)

Fig. 6-24● .file シンボルのエントリ

❖ .text のエントリ

オブジェクトファイルのセクション情報を格納します[41]。このエントリには，テキストセクションの情報が格納されます。

.text シンボルのエントリは，必ず補助エントリを使用します (Fig. 6-25)。各フィールドの意味は次のとおりです。

- *textsize* (テキストセクションのサイズ)
  テキストセクションのサイズを格納します。
- *lnum* (行番号テーブルのデータ数)
  使用した行番号テーブルのデータ数を格納します。

41).file シンボルのエントリ同様，複数のオブジェクトファイルをリンクした実行ファイルの場合は，リンクしたオブジェクトファイルの数だけこのエントリが存在します。これは，後述する .data，.bss シンボルについても同様です。

オフセット	内　容	
+$00 ⋮ +$07	*name*	(.text に固定)
+$08.l	(未使用)	
+$0C.w	*sect*	(必ず $0001 が入る)
+$0E.w	(未使用)	
+$10.b	*scl*	(必ず $78 が入る)
+$11.b	*auxent*	(必ず $01 が入る)
+$12.l	*textsize*	(テキストセクションのサイズ)
+$16.l	*lnum*	(行番号テーブルのデータ数)
+$1A ⋮ +$23	(未使用)	

Fig. 6-25 ● .text シンボルのエントリ

❖ .data のエントリ

このエントリには，データセクションの情報が格納されます。

`.data` シンボルのエントリは，必ず補助エントリを使用します (Fig. 6-26)。

フィールドの意味は次のとおりです。

- *datasize* (データセクションのサイズ)
  データセクションのサイズを格納します。

オフセット	内　容	
+$00 ⋮ +$07	*name*	(.data に固定)
+$08.l	*textsize*	
+$0C.w	*sect*	(必ず $0002 が入る)
+$0E.w	(未使用)	
+$10.b	*scl*	(必ず $78 が入る)
+$11.b	*auxent*	(必ず $01 が入る)
+$12.l	*datasize*	(データセクションのサイズ)
+$16 ⋮ +$23	(未使用)	

Fig. 6-26● .data シンボルのエントリ

❖ .bss のエントリ

このエントリには，ブロックストレージセクションの情報が格納されます。

.bss シンボルのエントリは，必ず補助エントリを使用します (Fig. 6-27)。

フィールドの意味は次のとおりです。

- *bsssize* (ブロックストレージセクションのサイズ)
  ブロックストレージセクションのサイズを格納します。

オフセット	内容	
+$00 ⋮ +$07	*name*	(.bss に固定)
+$08.l	*textsize* + *datasize*	
+$0C.w	*sect*	(必ず $0003 が入る)
+$0E.w	(未使用)	
+$10.b	*scl*	(必ず $78 が入る)
+$11.b	*auxent*	(必ず $01 が入る)
+$12.l	*bsssize*	(ブロックストレージセクションのサイズ)
+$16 ⋮ +$23	(未使用)	

Fig. 6-27 ● .bss シンボルのエントリ

◆ シンボルエントリの配列

シンボルエントリは，オブジェクトファイルや実行ファイルの中では Fig. 6-28 のような順で格納されています。

この図のタグ構造や関数構造は，Fig. 6-15（P.259）や Fig. 6-18（P.262）で説明したようなエントリの並びによって，それぞれの構造を表すものです。

リンカはリンク時に，これらのエントリを，リンクしたオブジェクトファイルの順に積み重ね，外部名の問題を解決した **extern** 変数名のエントリを，シンボルテーブルの末尾におくことで，実行ファイルのシンボルテーブルを作成します。

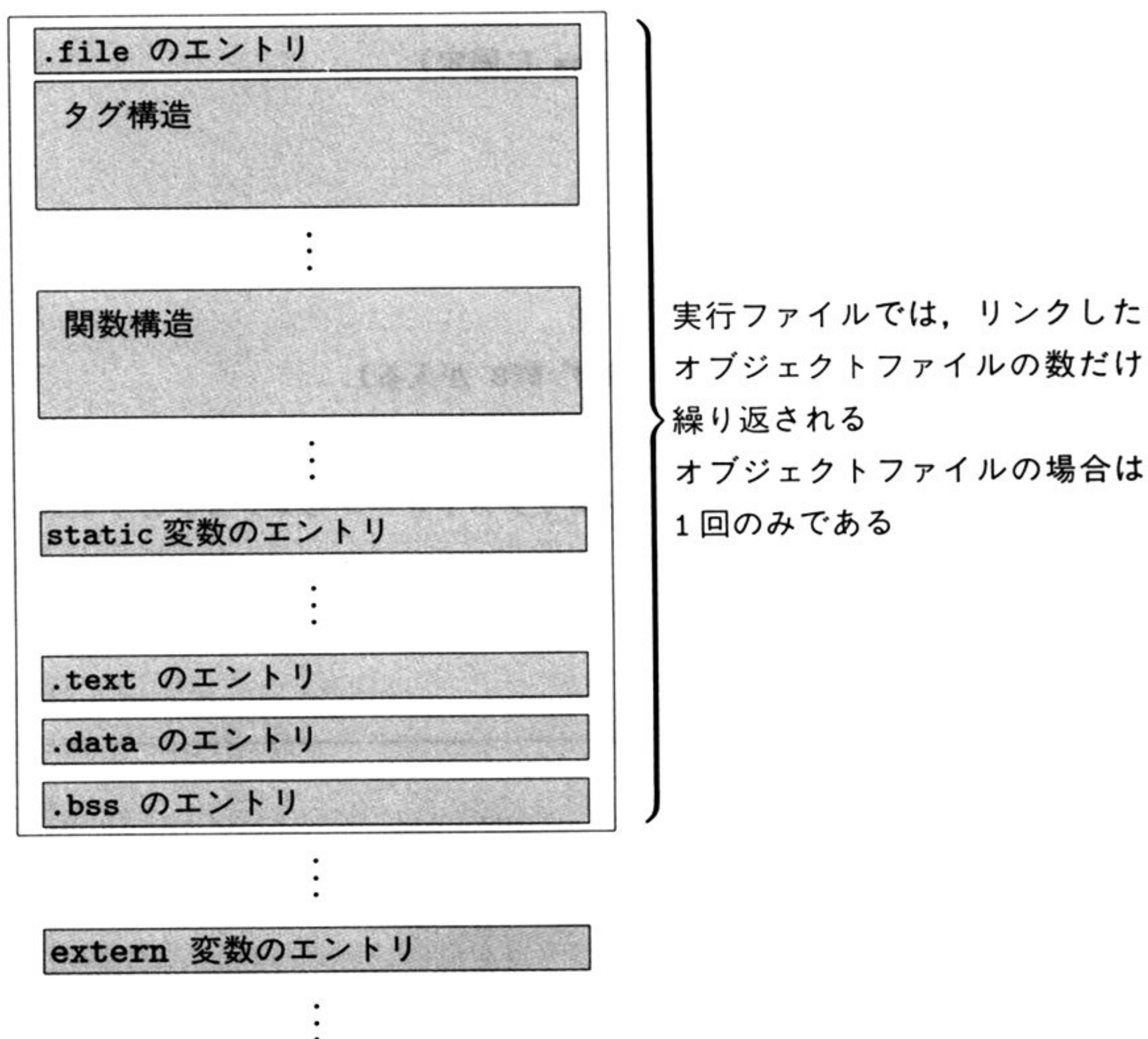



Fig. 6-28 ● シンボルエントリの配列

6.2.4 文字列テーブルのフォーマット

文字列テーブルは，単に 8 文字以上の長さをもったシンボル名を終了コード“$00”によって区切って並べたものです。各シンボルごとに，終了コードの後に偶数境界の調整が行われます。シンボルエントリは，文字列テーブル上のシンボル名をテーブル先頭からのオフセット値で参照します。

6.3 ················ ソースレベルデバッガの仕組み

C言語などで記述されたプログラムをマシンレベルデバッガでデバッグするには「コンパイラがソースプログラムをどのようなマシン命令に変換したのか」，「変数をどのメモリに割り当てたのか」，「ソースファイルの n 行目に対するマシン命令は何番地から何番地までなのか」ということをプログラマが知っている必要があります。これらについて知ることは，プログラマにとってかなりの負担です。

ソースレベルデバッガは，このようなソースレベルとマシンレベルでのめんどうな照らし合わせをサポートするユーザインタフェイスを，マシンレベルデバッガに拡張したものといえます。つまりソースレベルデバッガを実現するには，マシンレベルデバッグ機能とソースレベルからマシンレベルへ，またはマシンレベルからソースレベルへの問い合わせをする機能を実現すればよいことになります。

6.3.1 マシンレベルデバッグ機能

デバッグ対象プログラムの状態を調査したり，実行をコントロールするためには，次のような機能が必要です。

1. **メモリのデータの読み込み／書き込み**
2. **マシン命令のステップ実行**
3. **あるアドレスでプログラムの実行を停止する**[1)]

1)ブレークポイントのことです。

1. の機能は，指定したアドレスの内容にアクセスするだけですから，説明するまでもないでしょう。

2. の機能は，ハードウェアにステップ実行のための機能があれば問題はないのですが，その機能がなければブレークポイントを使って実現することができます。

3. の機能は，ソフトウェア割り込み[2)]を利用して実現することができます。

2)ソフトウェアブレークポイントといいます。

◆ ソフトウェアブレークポイント

問題となるソフトウェアブレークポイントを実現するための手順を，次に示します。

1. **ブレークポイントの設定**
   ブレークポイントを設定するアドレスの命令コードを一度退避して，そこに割り込み命令を書き込みます。

2. プログラムの実行とブレーク

プログラムが実行されていき，ブレークポイントを設定したアドレスにさしかかると，元の命令を置き換えた割り込み命令が実行されます。割り込み命令が実行されると，プロセッサの現在の内部状態をすべて保存し，デバッガに制御が戻ります。デバッガに制御が移ると，デバッガは割り込み命令を退避しておいた元の命令コードに戻して，プログラム実行の再開にそなえます。ここで注意しなければならないことがあります。それは，

割り込み命令を実行してから停止した

ということです。この状態で，ブレークポイントを設定したアドレスの割り込み命令から，退避しておいた命令コードに戻しただけでは，実行を再開させる場合，次の命令から実行されるか，または意味不明なコードを実行させることになってしまいます。つまり CPU のプログラムカウンタは，割り込み命令の次の命令のアドレスを指し示しているので，割り込み命令を実行する前の状態に戻す必要があります。またステータスレジスタも，停止直前の状態に戻しておかなければなりません。このことはデバッガに制御が移ったときに，プログラマがデバッガのコマンドを使い，停止したプログラムの状態を調べる際にも問題となります。

3. プログラム実行の再開

プログラム実行の再開には，割り込みからの復帰命令を使用して行います。復帰命令を実行すると，割り込み命令によってスタックに退避されていたプログラムカウンタやステータスレジスタを戻して，プログラムカウンタの指すアドレスに戻ります。しかし前述したように，プログラムカウンタは割り込み命令の次の命令を指し示しているため，スタックに退避されているプログラムカウンタのアドレスを，割り込み命令の長さ分だけ戻したアドレスに書き換えます。これでようやく，ブレークポイントが設定されていたアドレスから実行を再開させることができます。

6.3.2 ユーザインタフェイス

ソースレベルデバッグのユーザインタフェイスを構築するためには，ソースプログラムの情報が必要になります。これは，コンパイラがソースプログラムを解析して得た情報から生成されます。この情報のことをシンボル情報といいます。以降，説明のために C 言語をプログラミング言語として用います。

◆ シンボル情報

シンボル情報は，ソースプログラムとオブジェクトプログラムとの対応を行うためのものです。その中には「ソースプログラムで定義された手続き」や「変数の名前」,「型」,「記憶クラス」などと，これらをオブジェクトプログラムに結びつけるために必要となる「アドレス値」や「スタックフレームのオフセット」,「レ

ジスタへの対応情報」などから構成されます。次にソースレベルデバッグに必要な情報について，List 6-2 を例にして説明します。

List 6-2 ● sample.c

```
 1:  int sample (char *str, int ch)
 2:  {
 3:    int cnt;
 4:
 5:    cnt = 0;
 6:    while (*str)
 7:      {
 8:        if (*str == ch)
 9:          ++cnt;
10:        ++str;
11:      }
12:    return cnt;
13:  }
```

このソースプログラムには，次のような情報が必要です。

- ソースファイル名 “**sample.c**”
- 1 行目 … **int** を返す関数 “**sample**” の定義
  1 番目の引数 “**str**” は **char*** 型の変数
  2 番目の引数 “**ch**” は **int** 型の変数
  引数はスタックフレームに割り付けられるので，それぞれのオフセット位置
- 2 行目 … 関数ブロックの始まり
- **int** 型の変数 “**cnt**” の定義
  ローカル変数なので，割り当てられたスタックフレームの位置
- 5 行目 … 命令の開始アドレス
- 6 行目 … 命令の開始アドレス
- 7 行目 … 内部ブロックの始まり
- 8 行目 … 命令の開始アドレス
- 9 行目 … 命令の開始アドレス
- 10 行目 … 命令の開始アドレス
- 11 行目 … 内部ブロックの終わり
- 12 行目 … 命令の開始アドレス
- 13 行目 … 関数ブロックの終わり

❖ 型

型というのは，あるビット列 (またはあるバイト列) のデータをどのように解釈するのかについての情報のことです。たとえば，C 言語の基本データ型の **char** は 1 バイトの符号つき整数，**float** は 4 バイトの浮動小数点というように，型によってデータ長とデータの表現方法が異なります。ソースレベルデバッガは型の情報によって，変数の値を整数や浮動小数点などと区別して表示します。

C 言語では，**char**，**int** などの基本データ型と「～ の構造体へのポインタを返す関数」という複雑な型の両方を定義することができます。また構造体

などのように，いくつかの要素[3]をもった型も定義することができます。基本データ型は，プログラミング言語であらかじめ定義されているデータ型です。これらは，単純にコード化して表現することができます。しかし複雑な型を表現するには，ちょっとした工夫が必要です。たとえば **UNIX** の COFF では，基本データ型の情報に，

- ～ へのポインタ
- ～ の配列
- ～ を返す関数

といった情報を付加するようになっています。C 言語では，複雑な型でも基本データ型がもとになっているわけですから，これらの情報を付加するだけで通常の型は表現できます。

3)メンバといいます。

❖ スコープ

変数の中にはローカル変数，グローバル変数があります。そのためローカル変数では，

その変数が有効なスコープ内でだけ存在する

という，変数の通用範囲を考慮しなければなりません。C 言語などのブロック構造のプログラミング言語では，関数のブロック，その内部ブロック，そのまた内部ブロックというようにブロックの入れ子構造になっています。そして変数がどのブロックで定義されたかによって，そのシンボルのスコープが決定されます。つまり，どのブロックで変数が定義されているのかをデバッガが知らなければ，変数へのアクセスは正しく行えません。そのために，変数の通用範囲を決定する，それぞれのブロックの始まりと終わりの情報が必要になります。

❖ 格納場所

変数の内容にアクセスするためには，その変数のデータが格納されている場所[4]の情報が必要です。その情報は，静的変数と動的変数では次のように異なります。

4)メモリのアドレスのことです。

- **静的変数**
  静的変数はメモリの固定領域に割り付けられるので，メモリアドレスをシンボルの値とします。
- **動的変数**
  動的変数は通常スタック領域に割り当てられており，この領域のことをフレームと呼びます。関数が呼び出されるときには，その呼び出し用のフレームがスタック上に確保され，関数から戻るときにフレームは開放されます。したがって，動的変数は呼び出し時に確保されたフレーム上の位置[5]をシンボルの値とします。

5)フレームポインタからのオフセットになります。

❖ 実行文

コンパイルされたプログラムをソースレベルから実行制御するためには，ソースプログラムとオブジェクトプログラム間の対応情報が必要になります。その情報とは，次のような条件を満たすものです。

- **関数に対するマシン命令のオブジェクトプログラムのアドレス**
- **ソースプログラムの行番号に対するオブジェクトプログラムのアドレス**
- **あるマシン命令のアドレスに対する関数の割り出し**
- **あるマシン命令のアドレスに対するソースプログラムの行番号の割り出し**
- **あるソースプログラムの行番号(またはマシン命令のアドレス)に対するブロックの割り出し**
- **あるブロックの外側のブロックの割り出し**

❖ **関数に関する情報**

以下の情報は，関数へ渡された引数や関数ブロックのスコープを決定するために必要です。

- **関数の名称**
- **関数からの戻り値の型**
- **関数の呼び出し引数**
- **関数ブロックの先頭アドレス**
- **関数ブロックの終了アドレス**

❖ **行情報**

以下の情報は，ソースプログラム上の行番号とメモリにおかれた実行プログラムのアドレスとを結びつけるために必要です。

- **ソース行番号**
- **命令コード列の先頭アドレス**

❖ **ブロック**

識別子のスコープを決定するために必要な情報です。ブロック情報には次の3つのものが含まれています。

- **ブロック識別子**
- **ブロックの先頭アドレス**
- **ブロックの終了アドレス**

6.3.3 GDB の内部構造

このセクションでは，デバッグ対象プログラムを停止させたり，再実行させたりする機能であるプロセスをコントロールする方法について説明します。

◆ **プロセスのコントロール**

UNIX ではデバッガを使用するために PTRACE [6] というシステムコールが用意されています。このシステムコールは，指定したプロセスのメモリにアクセスしたり，実行をコントロールしたりすることができます。つまり，デバッガとして最低必要な機能は，このシステムコールを使うだけで実現できてしまいます。**GDB** も，この PTRACE によってデバッグ対象プログラムをコントロールするようになっています。

6)process trace を省略したものです。

◆ PTRACE

List 6-3，List 6-4 が，**Human68k** 版 **GDB** のために作成した PTRACE です。**Human68k** では，複数のプロセスを同時に実行させることができないため，PTRACE でコントロール可能なプロセスを 1 つに制限してあります。

List 6-3 では PT_SLEEP の処理が記述されていません。なぜならば，**UNIX** ではシステムコールの EXEC に PTRACE の処理が含まれているのに対して，**Human68k** 版 **GDB** では，PTRACE 専用の exec を別に用意しているからです。通常 **UNIX** の EXEC は，起動するプログラムをロード／実行します。しかし PTRACE でそのプロセスのトレースを宣言すると，プログラムをロードした後に実行を停止し，親プロセスに対してシグナルを出すようになります。親プロセスは，そのシグナルによってトレースするプログラムが実行される前に，ブレークポイントを設定することができます。つまり，デバッガからデバッグ対象プログラムを起動するときには，必ず PTRACE によってトレースされることになります。したがって **Human68k** 版 **GDB** では，**UNIX** の EXEC が PTRACE によるトレース宣言をしたときと同じ動作をする専用の exec を作ることで，これらの機能を実現しています。

PT_READ_I，PT_WRITE_I は，トレースするプロセスのメモリにアクセスするための機能です。**UNIX** の PTRACE ではメモリを参照する場合，プロセスの論理アドレスで指定しますが，**Human68k** 版 **GDB** では，プロセスに割り当てられたメモリ以外の領域へアクセスできるように，物理アドレスで指定するようにしてあります。

List 6-3 ● ptrace.c からの抜粋

```
 1:  int ptrace(int mode, int pid, unsigned long arg1, unsigned long arg2)
 2:  {
 3:    int result;
 4:
 5:    errno = 0;
 6:    switch (mode) {
 7:      case PT_SLEEP:
 8:        break;
 9:
10:      /* 指定のアドレスから 4 バイト読み込む */
11:      case PT_READ_I:
12:        if (main_memory_limit > arg1 || arg1 >= 0xc00000)
13:          result = B_LPEEK ((unsigned long *)arg1);
14:        else
15:          errno = 1;
16:        break;
17:
18:      /* チャイルドプロセスのレジスタの内容を読み込む */
19:      case PT_READ_U:
20:        result = *(unsigned long *)((unsigned long)inferior_regs + arg1);
21:        break;
22:
23:      /* 指定のアドレスへ 4 バイト書き込む */
24:      case PT_WRITE_I:
25:        if (main_memory_limit > arg1 || arg1 >= 0xc00000)
26:          *(unsigned long *)arg1 = arg2;
27:        else
28:          errno = 1;
```

```
29:         break;
30:
31:       /* チャイルドプロセスのレジスタへ書き込む */
32:       case PT_WRITE_U:
33:         *(unsigned long *)((unsigned long)inferior_regs + arg1) = arg2;
34:         break;
35:
36:       /* 停止しているチャイルドプロセスの実行を再開する */
37:       case PT_CONTINUE:
38:         ptrace_continue ();
39:         break;
40:
41:       /* チャイルドプロセスを終了させる */
42:       case PT_KILL:
43:         ptrace_kill ();
44:         break;
45:
46:       /* チャイルドプロセスの実行を再開し，マシン語命令の
47:          1 ステップを実行し終わったところで停止させる     */
48:       case PT_STEP:
49:         ptrace_step ();
50:         break;
51:
52:       default:
53:         errno = 1;
54:     }
55:     return result;
56: }
```

List 6-4 ● ptsub.s からの抜粋

```
 1: _ptrace_step:
 2:         bset     #7,_sr_reg+2
 3:         movea.l _pc_reg,a0
 4:         move.w  (a0),d0
 5:         and.w   #$f000,d0
 6:         cmp.w   #$f000,d0
 7:         beq     set_break_point
 8:         cmp.w   #$a000,d0
 9:         beq     set_break_point
10:         move.w  #0,indos
11: _ptrace_continue:
12:         movem.l d3-d7/a3-a7,_gdb_regs
13:         move.l  _inferior_pdb,-(sp)
14:         DOS     _SETPDB
15:         addq.l  #4,sp
16:
17:         move.l  #0,-(sp)
18:         DOS     _SUPER
19:         addq.l  #4,sp
20:         move.l  d0,_gdb_ssp
21:         bsr     change_vector_to_enable
22:
23:         movea.l _usp_reg,a0
24:         move.l  a0,usp
25:         move.l  _ssp_reg,sp
26:         move.w  _sr_reg+2,(sp)
27:         move.l  _pc_reg,2(sp)
28:         movem.l _inferior_regs,d0-d7/a0-a6
29:         rte
30:
31: _ptrace_kill:
```

```
32:          move.l  #abort_proc,_pc_reg
33:          bra     _ptrace_continue
```

PT_READ_U，PT_WRITE_U は，プロセスが停止したときに退避されるレジスタの値を参照または変更する機能です。退避されるレジスタの値は，4 バイトの配列になっています。それぞれのレジスタにアクセスするには，レジスタ番号を指定します。レジスタ番号は，Fig. 6-29 に示すとおりです。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
d0	d1	d2	d3	d4	d5	d6	d7	a0	a1	a2	a3	a4	a5	a6	a7	sr	pc

Fig. 6-29● レジスタ番号の対応

PT_CONTINUE は，指定したプロセス[7]の実行を再開する機能です。List 6-4 の _ptrace_continue がそのためのサブルーチンです。ここでは **GDB** のプロセスから，デバッグ対象プログラムのプロセスへ切り替えるための処理をしています。詳しくは「プロセス切り替え」(P.283) を参照してください。

7)ブレークポイントなどで停止しているプロセスを指します。

PT_KILL は，指定したプロセスを削除する機能ですが，**Human68k** には，プロセスを削除する機能がないために少々あらっぽい方法でプロセス削除を実現しています。あらかじめ，**GDB** の内部にアボート処理ルーチンを用意しておき，デバッグ対象プログラムの PC[8]をそのアボート処理ルーチンへ強制的に変更します。後は，PT_CONTINUE でプログラムの実行を再開すればアボート処理ルーチンが実行され，プロセスが削除されます。

8)プログラムカウンタのことです。

PT_STEP は，マシンレベルのステップ実行を行う機能です。マシンレベルのステップ実行には，68000 のトレース例外処理を使っています。トレース例外処理は，ステータスレジスタの T ビットをアサートしておくことで，マシン命令を 1 ステップ実行した後にソフトウェア割り込みを発生させます。そのソフトウェア割り込みは，シグナルとして **GDB** へ通知されます。

また DOS コールや SX システムコールは，内部をトレースしても意味がないため，一時的に，次の命令にブレークポイントを設定することでスキップするようにしてあります。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
T	0	S	0	0	I	I	I	0	0	0	X	N	Z	V	C

T → Trace Mode

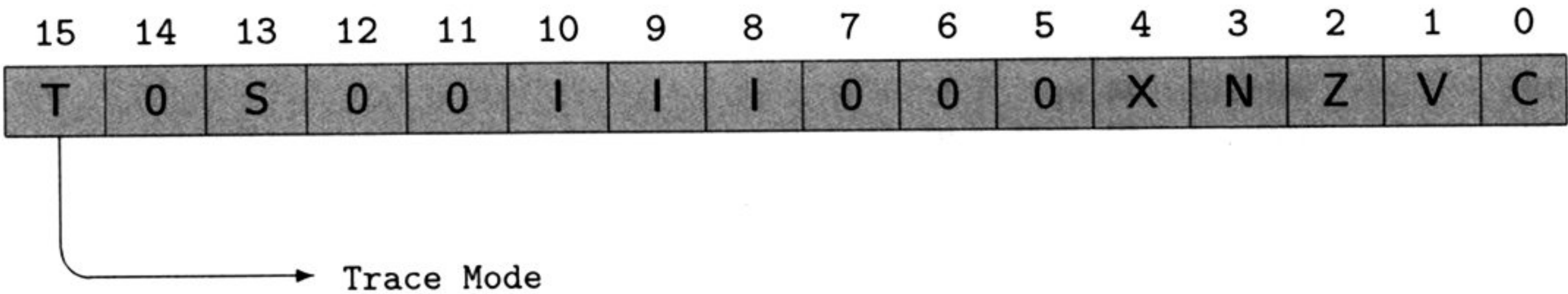



Fig. 6-30● SR(ステータスレジスタ)の構成

◆ ブレークポイント

ブレークポイントは，P.273 で説明したソフトウェアブレークポイントを使って実現しています。

プログラムの実行を停止させたいアドレスのマシン命令を，“trap #9” 例外処理命令に置き換えておくことで，ソフトウェア割り込みを発生させるようになっています。

“trap #9” 命令は “0x4e49” という 2 バイトのマシンコードですから，ブレークポイントを設定するアドレスから 2 バイトのマシンコードを退避しておき，そこに “trap #9” 命令を直接書き込みます。プログラムが “trap #9” を実行すると割り込みが発生し，設定しておいたサービスルーチンへ制御が移ります。そのサービスルーチンでは，レジスタの退避と GDB へのプロセス切り替えを行っています。List 6-5 の break_point_proc が，“trap #9” による割り込みによって実行されるサービスルーチンのエントリです。

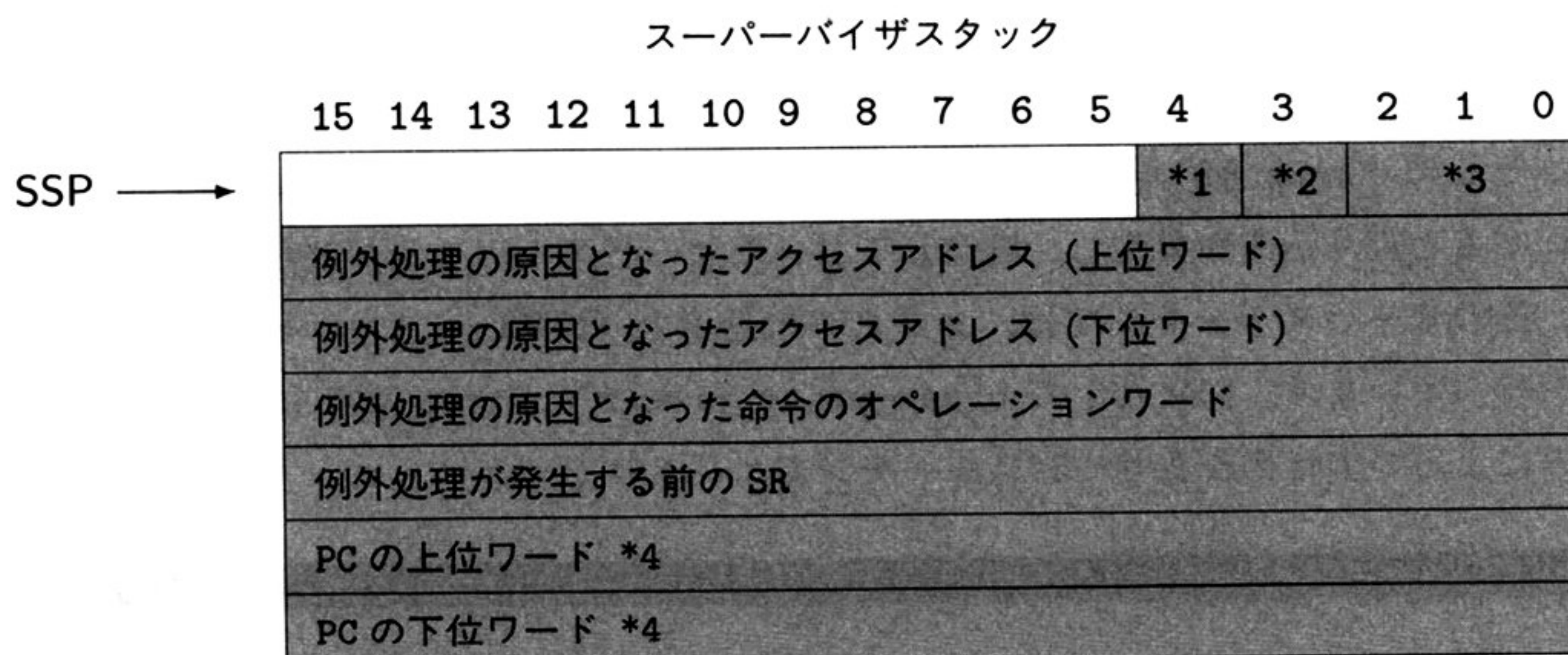
List 6-5 ● ptsub.s からの抜粋

```
 1: nmi_proc:
 2:         move.b  #$0c,$00e8e007
 3:         move.l  #$0000007f,_inferior_status
 4:         bra     save_reg
 5: bus_error_proc:
 6:         move.l  #$0002017f,_inferior_status
 7:         bra     save_reg
 8: addr_error_proc:
 9:         move.l  #$0003017f,_inferior_status
10:         bra     save_reg
11: illegal_inst_proc:
12:         move.l  #$0004027f,_inferior_status
13:         bra     save_reg
14: zero_div_proc:
15:         move.l  #$0005027f,_inferior_status
16:         bra     save_reg
17: chk_inst_proc:
18:         move.l  #$0006027f,_inferior_status
19:         bra     save_reg
20: trapv_inst_proc:
21:         move.l  #$0007027f,_inferior_status
22:         bra     save_reg
23: priv_proc:
24:         move.l  #$0008027f,_inferior_status
25:         bra     save_reg
26: break_point_proc:
27:         move.l  #$0001047f,_inferior_status
28: save_reg:
29:         movem.l d0-d7/a0-a6,_inferior_regs
30:         move.l  usp,a1
31:         move.l  a1,_usp_reg
32:         movea.l sp,a0
33:         cmpi.b  #1,_inferior_status+2
34:         bne     n_sigbus
35:         move.l  (a0),_except_info
36:         move.l  4(a0),_except_info+4
37:         addq.l  #8,a0
38:             :
39:             :
40:             :
```

◆ バスエラー／アドレスエラー

68000 はバスエラーまたはアドレスエラーが発生した場合，エラーの原因に関する情報をスタックに退避します。**Human68k** 版 **GDB** ではエラーが発生し，プログラムが停止したときに，その情報を表示するようになっています。

スーパーバイザスタック

	15 14 13 12 11 10 9 8 7 6 5	4	3	2 1 0
SSP →		*1	*2	*3
	例外処理の原因となったアクセスアドレス（上位ワード）			
	例外処理の原因となったアクセスアドレス（下位ワード）			
	例外処理の原因となった命令のオペレーションワード			
	例外処理が発生する前の SR			
	PC の上位ワード *4			
	PC の下位ワード *4			

*1　ビット 4:　0：ライトサイクルによって例外処理が発生した
　　　　　　　1：リードサイクルによって例外処理が発生した
*2　ビット 3:　0：命令をアクセスした
　　　　　　　1：命令以外をアクセスした
*3　ビット 2，1，0：FC2，FC1，FC0(ファンクションコード)の状態
*4　PC の値は，バスエラーまたはアドレスエラーを発生させた命令の次の命令が格納されているアドレスを保持しているとはかぎらない

※　SSP は 14 バイト (7 ワード) だけ更新される

Fig. 6-31● アドレスエラー(バスエラー)時のスタック

◆ シグナル

ブレークポイントまたはバスエラーなどの例外処理が実行されたとき，それらを識別する数値をシグナルとして **GDB** へ通知します。前出の List 6-5 が，例外処理によって実行されるサービスルーチンの一部を抜き出したものです。List 6-5 では例外処理が発生すると，それぞれのサービスルーチンへエントリし，プロセスが停止した原因[9]を “_inferior_status” に格納します。デバッガは，この “_inferior_status” の値を調べることで，デバッグ対象プログラムが停止した原因を知ることができます。

また “_inferior_status” には，シグナル番号のほかに，割り込み番号およびウエイトステータスも同時に格納されています。

9)シグナル番号のことです。

```
00 02 01 7f  ………… バスエラーで停止したときの値 (16進数)
 |  |  |  |
 |  |  |  └ウエイトステータス
 |  |  |      0x7f   プロセス停止
 |  |  |      0x00   プロセス終了
 |  |  |
 |  |  └シグナル番号
 |  |         0x00   インタラプトスイッチによる割り込み
 |  |         0x01   バスエラー, アドレスエラー
 |  |         0x02   その他のエラー        (不当命令など)
 |  |         0x04   ブレークポイント      (trap #9)
 |  |
 |  └例外処理番号
 |            0x00   NMI
 |            0x02   バスエラー
 |            0x03   アドレスエラー
 |            0x04   不当命令
 |            0x05   0による除算
 |            0x06   CHK命令
 |            0x07   TRAPV命令
 |            0x08   特権違反
```

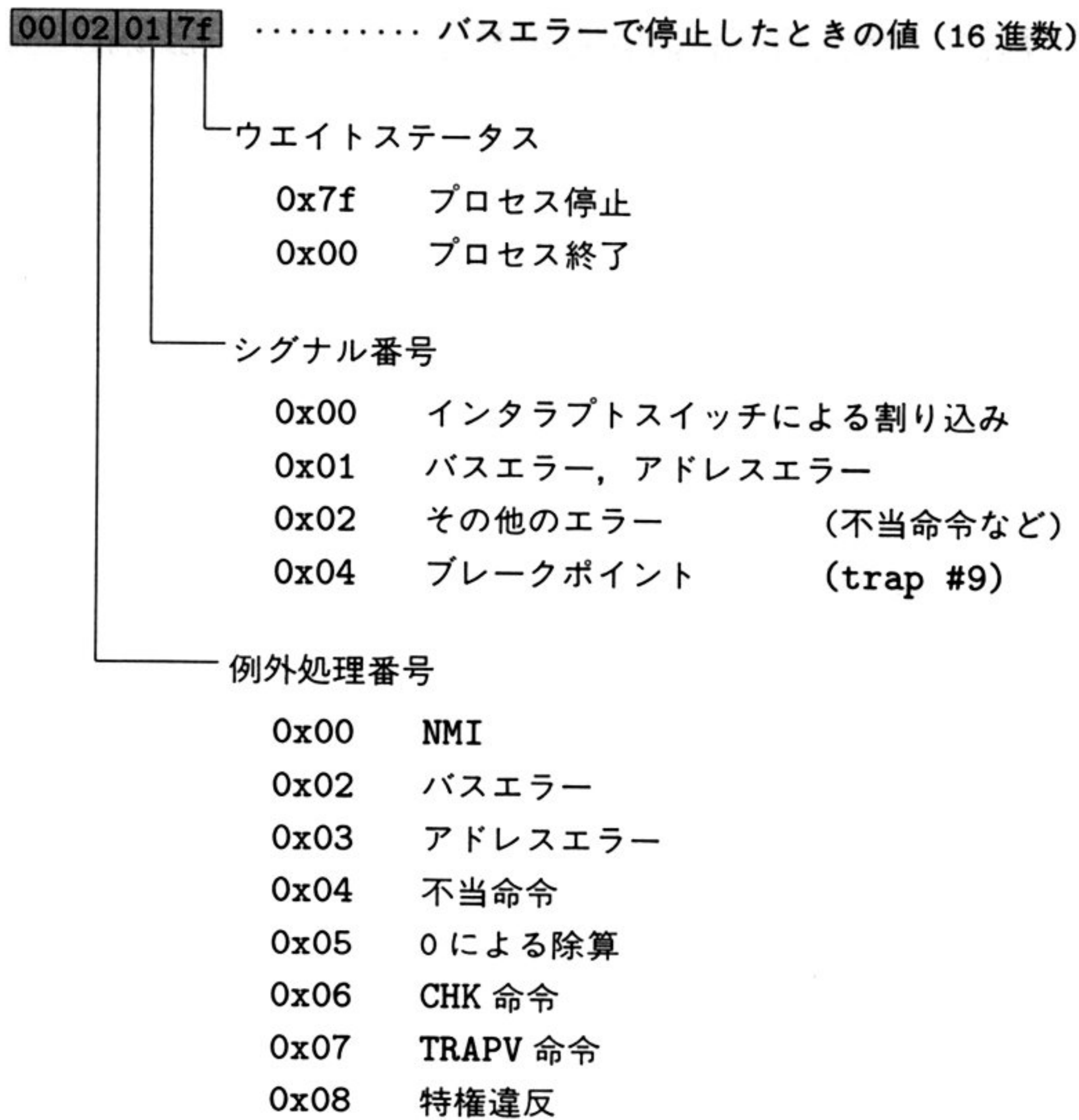



Fig. 6-32● _inferior_status の構成

◆ プロセス切り替え

デバッガを実現するには，デバッガとデバッグ対象プログラムの両方を同時に，または切り替えて実行しなければなりません。しかし **Human68k** のようなシングルタスク環境では同時に，2つのプロセスを実行することはできないため，デバッガのプロセスとデバッグするプログラムのプロセスとを切り替える必要があります。「プロセスを切り替える」というのは，デバッグするプログラムがブレークポイントで停止したときに，デバッガに制御を移したりデバッガのコマンドでデバッグするプログラムの実行を再開させるときのことです。

次に「ブレークポイントで停止し，デバッガに制御を移す場合」のプロセス切り替えの手順を示します。

1. デバッグするプログラムのプロセス状態[10]を退避する
2. OS にプロセス切り替えを通知する
3. デバッガに制御を移す[11]

10)レジスタのことです。

11)レジスタ，PC を復帰します。

次のプログラムリスト List 6-6 は，TRAP 命令によって割り込みが発生したときに実行される部分を抜き出したものです。プログラムの最後の部分はデバッガのプロセスに戻るための処理ですが，けっこうトリッキーなことをしているので説明しておきます。

List 6-6 ● ptsub.s からの抜粋

```
 1:          movem.l d0-d7/a0-a6,_inferior_regs      * レジスタの退避
 2:          move.l  usp,a1
 3:          move.l  a1,_usp_reg
 4:          movea.l sp,a0
 5:                 :
 6:                 :
 7:          move.l  a0,_ssp_reg
 8:          move.w  (a0),d0
 9:          bclr    #15,d0
10:          move.w  d0,_sr_reg+2
11:          move.l  2(a0),_pc_reg
12:          andi.w  #$f8ff,sr
13:                 :
14:                 :
15:          move.l  _gdb_pdb,-(sp)          * OS へプロセス切り替えの通知
16:          DOS     _SETPDB
17:          addq.l  #4,sp
18:          move.l  d0,_inferior_pdb
19:                 :
20:                 :
21:          movem.l _gdb_regs,d3-d7/a3-a6   * デバッガのレジスタ状態を復帰
22:          movea.l _gdb_ssp,sp
23:          movea.l _gdb_usp,a0
24:          move.l  (a0),-(sp)
25:          adda.l  #4,a0
26:          move.l  a0,usp
27:          move.w  #0,-(sp)
28:          move.l  #0,d0
29:          rte                             * デバッガのプロセスへ戻る
```

RTE 命令は TRAP 命令などによる割り込みによって，スタックに退避された SR[12] と PC を復帰して，割り込み前の処理にリターンするというものです。リターンするというよりも，スタックにおかれたアドレスにジャンプするという感じになります。

12)ステータスレジスタのことです。

この性質をうまく利用することで，プロセスを簡単に切り替えることができます。まず切り替えたいプロセスの実行を再開するアドレスと SR で，RTE 命令が受けつけるような形式で偽のスタックフレームを作成します。そして，スタックポインタを偽のスタックフレームに設定し，RTE 命令を実行するようにします。すると CPU は，まんまとだまされて偽のスタックフレームを取り込み，他のプロセスを実行し始めることができます。

◆ デバッグ対象プログラムの起動

デバッグ対象プログラムは，GDB プロセスのチャイルドプロセスとして起動しますが，プログラムの実行前にブレークポイントを設定する必要があるため，

- **ロード**
- **実行**

の2段階に分けて行われています。List 6-7 は，デバッグ対象プログラムをチャイルドプロセスとして起動するための処理の最も低レベルな部分を抜き出したものです。

List 6-7 ● ptsub.s からの抜粋

```
 1:  __exec:
 2:          move.l  4(sp),file
 3:          move.l  8(sp),comline
 4:          movem.l d3-d7/a3-a7,_gdb_regs    * レジスタの退避
 5:          movea.l sp,a6
 6:
 7:          move.l  #0,-(sp)                 * スーパーバイザへ切り替え
 8:          DOS     _SUPER
 9:          addq.l  #4,sp
10:          move.l  d0,_gdb_ssp              * gdb プロセスの ssp を得る
11:                  :
12:                  :
13:          割り込みベクタの設定
14:                  :
15:                  :
16:          pea     _inferior_ssp_start      * チャイルドプロセスの ssp の
17:          DOS     _SUPER                   * 設定，ユーザモードへ切り替え
18:          addq.l  #4,sp
19:          lea     _inferior_usp_start,sp   * チャイルドプロセスの sp 設定
20:
21:          move.l  #0,_sr_reg
22:          move.w  #0,indos
23:
24:          pea     0                        * プログラムのロード
25:          move.l  comline,-(sp)
26:          move.l  file,-(sp)
27:          move.w  #1,-(sp)
28:          DOS     _EXEC
29:          lea     14(sp),sp
30:          tst.l   d0
31:          blt     return_to_gdb
32:          adda.l  #$100,a0
33:          move.l  a0,_load_addr            * ロードしたアドレスを退避
34:
35:          trap    #9                       * デバッガへ制御を移す
36:
37:          move.l  d0,-(sp)                 * 実行
38:          move.w  #4,-(sp)
39:          DOS     _EXEC
40:          lea     6(sp),sp
41:                  :
42:                  :
43:                  :
```

プログラムのロードには，**Human68k** の DOS コール“EXEC”を使っています。この“EXEC”には，プログラムをロード／実行させる機能がありますが，引数にコントロールモードを指定すると，ロードのみ行わせることができます。戻り値は，プログラムの実行アドレスとメモリ管理ポインタです。プログラムの実行アドレスは，次にプログラムの実行を開始させるときに使います。またメモリ管理ポインタによって，プログラムがロードされたアドレスを知ることができるので，デバッガからデバッグ対象プログラムにアクセスするために必要な情報です。

次に最初のブレークポイントを設定するために，デバッガに制御を移します。List 6-7 の 35 行目 の“**trap #9**”がそのための命令です。これはブレークポイ

ントの命令と同じものです。**GDB** は，この“`trap #9`”命令によって，プロセスが戻ってきたときに最初のブレークポイントを設定し，再びデバッグするプログラムの実行を再開させます。

◆ メモリ管理

GDB はメモリを動的に確保するため，起動時に OS からプロセスに与えられたメモリでは不足する場合があります。通常のアプリケーションであれば，`malloc` できなくなった時点で，ヒープを拡張することになります。しかし **GDB** のチャイルドプロセスとして起動されるデバッグ対象プログラムの場合，**GDB** プロセスの次のメモリにロードされるため，**GDB** のプロセスは，ヒープを拡張できなくなります。これは，OS に仮想記憶機構があれば問題ないのですが[13)]，現在の **X68000** にはその機能がありません。このままでは，**GDB** を **Human68k** で実現できなくなってしまいます。そこで現在の **Human68k** 版 **GDB** では，あらかじめ **GDB** プロセスのためにメモリを確保してから，デバッグ対象プログラムを起動するといった，メモリ効率を無視した方法を使用しています。

13)X68000 では，OS だけの問題ではないのですが・・・。

GDB プロセスのためにあらかじめ確保する領域は，**GDB** の起動時オプション“`-mem=SIZE`”によって，ユーザが指定したサイズの領域です。**GDB** は，この領域の範囲以内でヒープを拡張することができます。

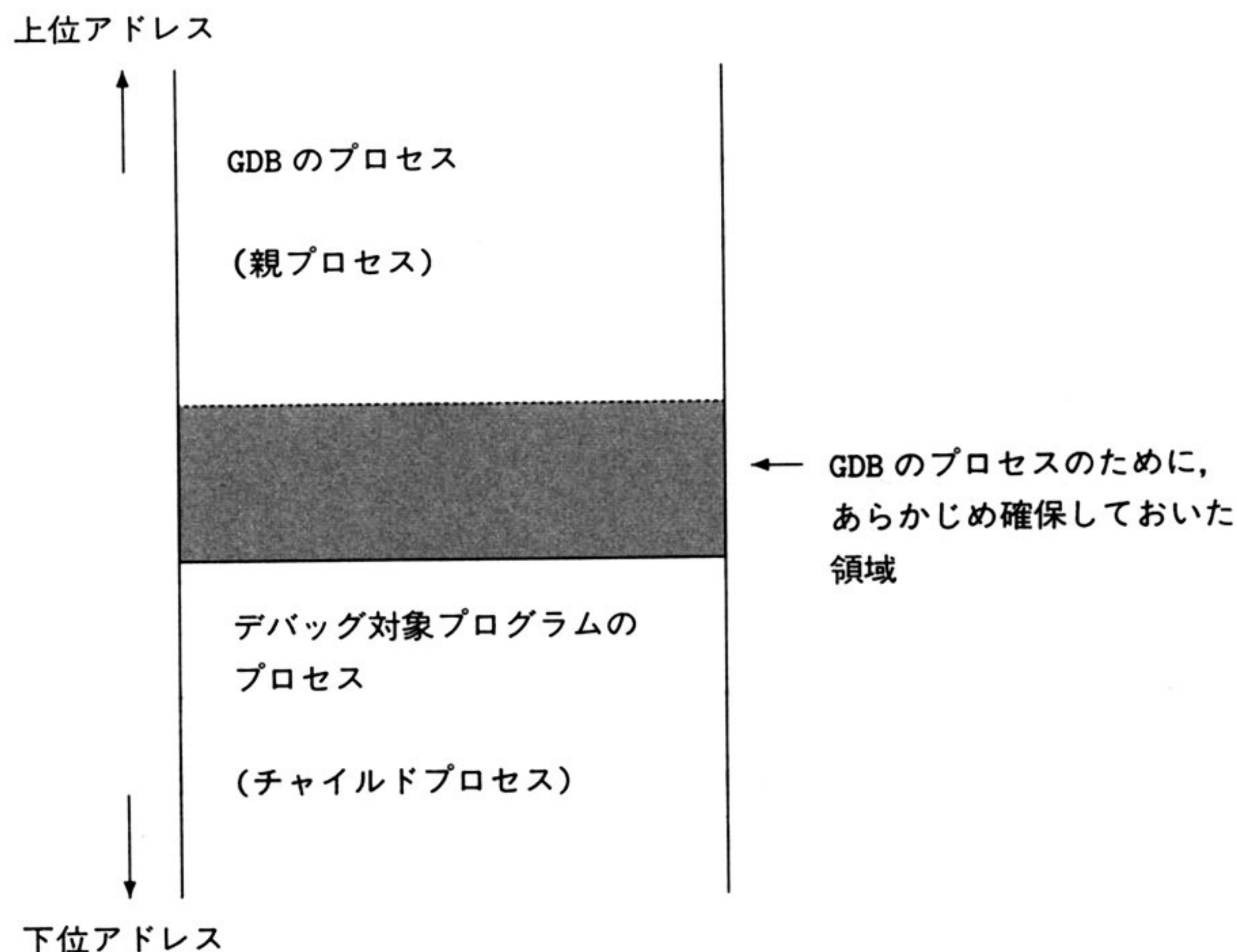



Fig. 6-33● メモリ管理状態

参考文献

Using GDB FSF Richard M. Stallman and Roland H. Pesch

CQ 出版社　インタフェイス '91-8

6.4 ……………………………… アドレス形式の表記

68000 CPU では，各実行命令の操作対象となる実効アドレスをオペランドとして指定する必要がありますが，この方法をアドレス形式といいます。このセクションでは，アドレス形式の種類とアセンブリ言語での表記の方法について説明します。レジスタ名については，Table 3-1(P.83) を参照してください。

6.4.1　データレジスタ直接形式

操作対象は指定したデータレジスタであり，命令は指定したレジスタの内容に対して直接実行されます[1)]。

書式：　Dn　$(n = 0 \sim 7)$
　　　　Rn　$(n = 0 \sim 7)$

1)R0～R7 は，それぞれ D0～D7 に対応します。

6.4.2　アドレスレジスタ直接形式

操作対象は指定したアドレスレジスタであり，命令は指定したレジスタの内容に対して直接実行されます[2)]。

書式：　An　$(n = 0 \sim 7)$
　　　　Rn　$(n = 8 \sim 15)$

2)R8～R15 は，それぞれ A0～A7 に対応します。

6.4.3　アドレスレジスタ間接形式

操作対象は，指定したアドレスレジスタの内容によってポイントされるメモリ内のデータです。

書式：　(An)　$(n = 0 \sim 7)$
　　　　(Rn)　$(n = 8 \sim 15)$

6.4.4 ポストインクリメント・アドレスレジスタ間接形式

操作対象は，指定したアドレスレジスタの内容によってポイントされるメモリ内のデータです。オペランドに対して命令が実行された後に，指定のアドレスレジスタに命令のサイズ[3]が加算されます。

3)'.b' なら 1，'.w' なら 2，'.l' なら 4 になります。

書式： (An)+ (n = 0 ∼ 7)
(Rn)+ (n = 8 ∼ 15)

6.4.5 プリデクリメント・アドレスレジスタ間接形式

操作対象は，指定したアドレスレジスタの内容によってポイントされるメモリ内のデータです。オペランドに対する命令の実行に先立って，指定のアドレスレジスタから命令のサイズ[4]が減算されます。

4)'.b' なら 1，'.w' なら 2，'.l' なら 4 になります。

書式： −(An) (n = 0 ∼ 7)
−(Rn) (n = 8 ∼ 15)

6.4.6 ディスプレースメントつきアドレスレジスタ間接形式

指定したアドレスレジスタの内容と 16 ビットのディスプレースメント値 (−32768 ∼ 32767) を，32 ビットに符号拡張した値を加算した値が実効アドレスとなります。命令の操作は，この実効アドレスのメモリの内容に対して行われます。

書式： d(An) (d = −32768 ∼ 32767)
d(Rn) (d = −32768 ∼ 32767)
HAS 拡張 (d,An)，(d,Rn) という表記も使用できます。

6.4.7 インデックスつきアドレスレジスタ間接形式

指定したアドレスレジスタの内容と，8 ビットのディスプレースメント値 (−128 ∼ 127) を 32 ビットに符号拡張した値，そしてインデックスとして指定したレジスタ[5]の内容を加算した値が実効アドレスとなります。命令は，この実効アドレスのメモリの内容に対して行われます。

5)データレジスタまたはアドレスレジスタです。

インデックスとして指定するレジスタには，ワードサイズまたはロングワードサイズが使用できます。ワードサイズを指定した場合には，指定したレジスタの下位ワードの値を，32 ビットに符号拡張してから使用されます。

書式：　d(An,Rn.w)　　($d = -128 \sim 127$)
d(An,Rn.l)　　($d = -128 \sim 127$)
HAS 拡張 (d,An,Rn.w)，(d,An,Rn.l) という表記も使用できます。

6.4.8　絶対ショートアドレス形式

実効アドレスには，オペランドに指定した 16 ビットの絶対アドレスを 32 ビットに符号拡張した値が使用されます。したがって，指定可能なアドレスの範囲は $00000000 から $00007FFF，および $FFFF8000 から $FFFFFFFF にかぎられます。

書式：　< 式 > .w

6.4.9　絶対ロングアドレス形式

実効アドレスには，オペランドに指定した 32 ビットの絶対アドレスが直接使用されます。

書式：　< 式 > .l[6]

6)'.l' は省略することができます。また，アセンブルの際のコマンドライン上で '-a' スイッチ（絶対ショートアドレス形式対応モードの指定）を指定すると，<式>の値によって絶対ショートアドレス形式と絶対ロングアドレス形式を自動的に選択します。

6.4.10　ディスプレースメントつきプログラムカウンタ相対形式

プログラムカウンタの内容と，16 ビットのディスプレースメント値 ($-32768 \sim 32767$) を 32 ビットに符号拡張した値を加算した値が実効アドレスとなります。命令は，この実効アドレスのメモリの内容に対して行われます。

書式：　d(PC)　　($d = -32768 \sim 32767$)
HAS 拡張 (d,PC) という表記も使用できます。

6.4.11　インデックスつきプログラムカウンタ相対形式

プログラムカウンタの内容と，8 ビットのディスプレースメント値 ($-128 \sim 127$) を 32 ビットに符号拡張した値，そしてインデックスとして指定したレジスタ[7]の内容を加算した値が実効アドレスとなります。命令の操作は，この実効アドレスのメモリの内容に対して行われます。

7)データレジスタまたはアドレスレジスタです。

インデックスとして指定するレジスタには，ワードサイズまたはロングワードサイズが使用できます。ワードを指定した場合には，指定したレジスタの下位ワードの値を，32 ビットに符号拡張してから使用されます。

書式：　d(PC,Rn.w)　　($d = -128 \sim 127$)
　　　　d(PC,Rn.l)　　($d = -128 \sim 127$)
　　　　HAS 拡張 (d,PC,Rn.w)，(d,PC,Rn.l) という表記も使用できます。

6.4.12　イミディエイト形式

命令の操作の対象は，オペランドに指定した値そのものとなります。指定する値の先頭には，イミディエイト形式であることを示すために“#”をおきます。

書式：　# < 式 >

6.4.13　クイックイミディエイト形式

処理の内容はイミディエイト形式と変わりませんが，値の範囲が限定される代わりに高速に実行できる機械語コードが生成されます。

この形式が使用できるのは，次の 3 つの命令です。

- MOVEQ
- ADDQ
- SUBQ

書式：　# < 式 >[8)]

8)値の範囲は命令によって変わります。

6.4.14　SR／CCR 形式

命令の操作の対象は，ステータスレジスタ (SR) またはコンディションコードレジスタ[9)]の内容です。

この形式が使用できるのは，以下の命令です。

9)CCR…SR レジスタの下位バイトです。

- ANDI to SR
- ANDI to CCR
- EORI to SR
- EORI to CCR
- ORI to SR
- ORI to CCR
- MOVE to SR

- MOVE to CCR
- MOVE from SR

書式：　SR

　　　　CCR

Chapter 7

Appendix B

本書および付属ディスクに含まれるすべてのプログラムは，次ページからの「GNU 一般公有使用許諾書」を適用しています。公的・私的に関わらず，これらのプログラムを使用する場合は，必ず「GNU 一般公有使用許諾書」全文をお読みください。

7.1………「GNU 一般公有使用許諾書」全文

注意

英文文書(GNU General Public Licence)を正式文書とする。この和文文書は弁護士の意見を採り入れて，できるだけ正確に英文文書を翻訳したものであるが，法律的に有効な契約書ではない。

和文文書自体の再配布に関して

いかなる媒体でも次の条件がすべて満たされている場合に限り，本和文文書をそのまま複写し配布することを許可する。また，あなたは第三者に対して本許可告知と同一の許可を与える場合に限り，再配布することが許可されています。

- 受領，配布されたコピーに著作権表示および本許諾告知が前もって載せられていること。
- コピーの受領者がさらに再配布する場合，その配布者が本告知と同じ許可を与えていること。
- 和文文書の本文を改変しないこと。

GNU 一般公有使用許諾書

1989年2月，バージョン1

Copyright ©1989 Free Software Foundation, Inc.

675 Mass Ave, Cambridge, MA 02139, USA

何人も，以下の内容を変更しないでそのまま複写する場合に限り，本使用許諾書を複製したり頒布することができます。

はじめに

ソフトウェア会社の使用許諾契約書は，多くの場合，その企業の意のままにユーザを縛ろうとしています。それに対して，我々の一般公有使用許諾は，フリー・ソフトウェアを共有したり変更する自由をユーザに保証するためのもの，即ちフリー・ソフトウェアがそのユーザ全てにとってフリーであることを保証するためのものです。本使用許諾は，Free Software Foundation のソフトウェアに適用されるだけでなく，プログラムの作成者が本使用許諾に依るとした場合のそのプログラムにも適用することができます。また，ユーザのプログラムのためにも利用することができます。

我々がフリー・ソフトウェアについて言う場合は自由のことに言及しているのであって，価格のことではありません。特に，一般公有使用許諾の各条項は，次の事柄を確実に実現することを目的として立案されています。

- フリー・ソフトウェアの複製物を自由に頒布したり販売できること。
- 希望しさえすればソース・コードを現実に入手できるか，あるいはその入手が可能であること。
- 入手したソフトウェアを変更したり，新しいフリー・プログラムの一部として使用できること。
- 以上の各内容を行なうことができるということをユーザ自身が知っていること。

このようなユーザの権利を守るために，我々は，何人もこれらの権利を否定したり，あるいは放棄するようにユーザに求めることはできないという制限条項を設ける必要があります。これらの制限条項は，ユーザが，フリー・ソフトウェアの複製物を頒布したり変更しようとする場合には，そのユーザ自身が守るべき義務ともなります。

例えば，あなたがフリー・ソフトウェアの複製物を頒布する場合，有償か無償かにかかわらず，あなたは自分の持っている権利を全て相手に与えなければなりません。あなたは，相手もまたソース・コードを受け取ったり入手できるということを認めなければなりません。さらにあなたは，相手が受領者へそれらの権利を持っているということを，その相手に知らせなければなりません。

我々は次の二つの方法でユーザの権利を守ります。(1) ソフトウェアに著作権を主張し，(2) 本使用許諾の条項の下でソフトウェアを複製・頒布・変更する権利をユーザに与えます。

また，各作成者や我々自身を守るために，本フリー・ソフトウェアが無保証であることを全ての人々が了解している必要があります。更に，他の誰かによって変更されたソフトウェアが頒布された場合，受領者はそのソフトウェアがオリジナル・バージョンではないということを知らされる必要があります。それは，他人の関与によって原開発者に対する評価が影響されないようにするためです。

複写・頒布・変更に対する正確な条項と条件を次に示します。

GNU 一般公有使用許諾の下での複製，頒布，変更に関する条項と条件

1. 本使用許諾は，一般公有使用許諾の各条項に従って頒布されるという著作権者からの告知文が表示されているプログラムやその他の作成物に適用されます。以下において「プログラム」とは，そのようなプログラムや作成物を指すものとし，また,「プログラム生成物」とは，上述した「プログラム」自身，及びその「プログラム」の全部又は一部の作成を，そのまま又は変更して内部に組み込んだ作成物を意味するものとします。本使用許諾によって許諾を受ける者を「あなた」と呼びます。

2. あなたは，どのような媒体上へ複製しようとする場合であっても，入手した「プログラム」のソース・コードをそのままの内容で複写した上で適正な著

作権表示と保証の放棄と明確，且つ適正に付記する場合に限り，複製又は頒布することができます。その場合，本一般公有使用許諾及び無保証に関する記載部分は，全て元のままの形で表示して下さい。また,「プログラム」の頒布先に対しては,「プログラム」と共に本一般公有使用許諾書のコピーを渡して下さい。複製物を引き渡す際の実費は請求することができます。

3. 次の各条件を満たしている限り，あなたは,「プログラム」又はその一部分を，変更することができます。更に，上記第2項を満たせば，その変更版を複製したり頒布することもできます。

   - ファイルを変更した旨とその変更日とを，変更したファイル上に明確に表示すること。
   - 変更したか否かを問わず，凡そ「プログラム」又はその一部分を内部に組み込んでいる作成物を頒布する場合には，本一般公有使用許諾の条項に従って無償で使用許諾すること。但し，頒布先の全て又はその一部の者に対して，あなたが独自に保証することは構いません。
   - 変更したプログラムが実行時に通常の対話的な方法でコマンドを読むようになっているとすれば，最も単純，且つ普通の方法で対話的にそのプログラムを実行する時に，次の内容を示す文言がプリント・アウトされるか，或は画面に表示されること。
     - 適切な著作権表示。
     - 無保障であること (あなたが独自に保証する場合は，その旨)。
     - 頒布を受ける者も，本一般公有使用許諾と同一の条項に従ってそのプログラムを再頒布できること。
     - 頒布を受ける者が本一般公有使用許諾書の写しを回覧する方法。
   - 複製物の譲渡に要する実費は請求できること。また，あなた独自の保証を行なう場合はそれを有償とすることができること。

   本「プログラム」(または，その派生物) と他の別個のプログラムとを，保管や頒布のために同一の媒体上にまとめて記録したとしても，本使用許諾の条項は他の別個のプログラムには適用されません。

4. あなたは，以下のうちいずれか1つを満たす限り，上記第2項及び第3項に従って「プログラム」(または，上記の条項2のもとのその一部分あるいはその派生物) をオブジェクト・コードまたは実行可能な形式で複製および頒布することができます。

   - 対応する機械読み取り可能なソース・コード一式を一緒に引き渡すこと。その場合，そのソース・コードの引き渡しは上記第2項及び第3項に従って行なわれること。
   - 少なくとも3年間の有効期間を定め，且つその期間内であれば対応する機械読み取り可能なソース・コード一式を無償で (ただし，少額の頒布実費は請求できる) 提供する旨，及びその場合には上記第2項及び第3項に従って提供される旨を記載した書面を，一緒に引き渡すこと。

❖ 対応するソース・コードを入手できる所について，あなたが得た情報を提供すること(この選択肢は，営利を目的としない頒布であって，且つあなたがオブジェクト・コードあるいは実行可能形式のプログラムしか入手していない場合にのみ適用される選択項目です。)

上記においてソース・コードとは，変更作業に適した記述形式を指します。また，実行可能形式のファイルに対応するソース・コード一式とは，それに含まれる全モジュールに対応する全てのソース・コードを指しますが，例外として，実行可能なファイルが動作するオペレーティング・システムに付随する標準ライブラリのモジュールのソース・コードやそのオペレーティング・システムに付随する定義ファイルのソース・コードを含ませる必要はありません。

5. 本一般公有使用許諾が明示的に許諾している場合を除き，あなたは，「プログラム」を複製，変更，サブライセンス，頒布，譲渡することができません。本使用許諾に従わずに「プログラム」を複製，変更，サブライセンス，頒布，譲渡しようとする行為は，それ自体が無効であり，且つ，本使用許諾があなたに許諾している「プログラム」の使用権限を自動的に消滅させます。その場合，本使用許諾に従ってあなたから複製物やその使用許諾を得ている第三者は，本使用許諾に完全に従っている限り，引続き有効な使用権限を持つものとします。

6. あなたが「プログラム」(あるいはその「プログラム生成物」)の複製，頒布，変更を行なえば，それ自体で，それらの各行為を行なう権利と，本使用許諾が定める全ての条項に従うことを，あなたが受け入れたものとみなします。

7. あなたが「プログラム」(あるいはその「プログラム生成物」)を再頒布すると自動的に，その受領者は，元の使用許諾者から，本使用許諾の条項に従って「プログラム」を複製，頒布，変項することを内容とする使用許諾を受けたものとします。あなたは，受領者に許諾された権利の行使について，更に制約を加えることはできません。

8. Free Software Foundation は随時，一般公有使用許諾の改訂版，又は新版を公表することがあります。そのような新しいバージョンは，現行のバージョンと基本的に変わるところはありませんが，新しい問題や懸案事項に対応するために細部では異なるかもしれません。
各バージョンは，バージョン番号によって区別します。「プログラム」中に使用許諾のバージョン番号の指定がある場合は，その指定されたバージョンか，又はその後に Free Software Foundation から公表されているいずれかのバージョンから1つを選択して，その条項と条件に従って下さい。「プログラム」中に使用許諾のバージョン番号の指定がない場合は Free Software Foundation が公表したどのバージョンでも選択することができます。

9. 「プログラム」の一部を頒布条件の異なる他のフリー・プログラムに組み込みたい場合は，その開発者に書面で許可を求めて下さい。Free Software Foundation が著作権を持っているソフトウェアについては，Free Software Foundation へ書面を提出して下さい。このような場合へ対応するために我々

は例外的処理をすることもありますが，その判断基準となるのは，次の2つの目標の実現に合致するか否かという点です。即ち，一つは我々のフリー・ソフトウェアの全ての派生物をフリーな状態に保つことであり，もう一つはソフトウェアの共有と再利用とを広く促進させることです。

無保証

10. 「プログラム」は無償で使用許諾されますので，適用法令の範囲内で，「プログラム」の保証は一切ありません。著作権者やその他の第三者は全く無保証で「そのまま」の状態で，且つ，明示か暗黙であるかを問わず一切の保証をつけないで提供するものとします。ここでいう保証とは，市場性や特定目的適合性についての暗黙の保証も含まれますが，それに限定されるものではありません。「プログラム」の品質や性能に関する全てのリスクはあなたが負うものとします。「プログラム」に欠陥があるとわかった場合，それに伴う一切の派生費用や修理・訂正に要する費用は全てあなたの負担とします。

11. 適用法令の定め，又は書面による合意がある場合を除き，著作権者や上記許諾を受けて「プログラム」の変更・再頒布を為し得る第三者は，「プログラム」を使用したこと，または使用できないことに起因する一切の損害について何らの責任も負いません。著作権者や前記の第三者が，そのような損害の発生する可能性について知らされていた場合でも同様です。なお，ここでいう損害には，通常，損害，特別損害，偶発損害，間接損害が含まれます(データの消失，又はその正確さの喪失，あなたや第三者が被った損失，他のプログラムとのインタフェースの不適合化，等も含まれますが，これに限定されるものではありません)。

──────────────(以上)──────────────

付録:　あなたの新しいプログラムにこれらの条項を適用する方法

あなたが新しくプログラムを作成し，それを人間性に一番則った方法で活用したい場合は，プログラムをフリー・ソフトウェアにして，全ての人々が以上の各条項に従ってこれを再頒布や変更をすることができるようにするのが最良の方法です。

そうするためには，プログラムに以下の表示をして下さい。その場合，無保証であるということを最も効果的に伝えるために，ソース・ファイルの冒頭にその全文を表示すれば最も安全ですが，その他の方法で表示する場合でも，「著作権表示」と全文を読み出す為のアドレスへのポインタだけはファイル上に表示しておいて下さい。

```
<プログラム名とどんな動作をするものかについての簡単な説明の行>
Copyright (C) 19**年, 著作権者名

本プログラムはフリー・ソフトウェアです。あなたは，Free Software Foundation が公表したGNU 一般公有使用許諾の「バージョン 1」あるいはそれ以降の各バージョンの中からいずれかを選択し，そのバージョンが定める条項に従って本プログラムを再頒布または変更することができます。
本プログラムは有用とは思いますが，頒布にあたっては，市場性及び特定目的適合性についての暗黙の保証を含めて，いかなる保証も行ないません。詳細についてはGNU 一般公有使用許諾書をお読み下さい。
あなたは，本プログラムと一緒にGNU 一般公有使用許諾の写しを受け取っているはずです。そうでない場合は，Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA へ手紙を書いて下さい。
また，ユーザが電子メイルや書信であなたと連絡をとる方法についての情報も書き添えて下さい。
```

プログラムが対話的に動作する場合は，対話モードで起動した時に次のような短い告知文が表示されようにして下さい。:

```
Gnomovision バージョン 69, Copyright (C) 19**年, 著作権者名

Gnomovision は完全に無保証です。詳細は 'show w' とタイプして下さい。これはフリー・ソフトウェアなので，特定の条件の下でこれを再頒布することができます。詳細は 'show c' とタイプして下さい。
```

上記の 'show w' や 'show c' は各々，一般公有使用許諾の関連する部分を表示するコマンドを指します。もちろん，あなたが使うこれらのコマンドは 'show w' や 'show c' といった呼び名でなくても構いません。更に，それらのコマンドはあなたのプログラムに合わせる為に，マウスでクリックしたりメニュー形式にしたりすることもできます。

また，必要と認めた場合には，あなたの雇い主 (あなたがプログラマとして働いている場合) や在籍する学校から，そのプログラムに対する「著作権放棄」を認めた署名入りの書面を入手して下さい。ここにその文例を載せます。名前は変えて下さい。

```
Yoyodyne, Inc. は, James Hacker が開発したプログラム
'Gnomovision' (コンパイラを起動してアセンブラにつなげるプログラム)
についての著作権法上の全ての権利を放棄する。

< TY COON の署名>, 1 April 1989
Ty Coon, 副社長
```

これで手続きは終りです!

INDEX

記号

A

B

C

D

E

L

M

N

O

P

Q

R

S

あとがき

「フリーウェア開発キットの発売」などという，まるで思いもよらなかった話を突然もちかけられてから，気がつくとこうして形になるところまで来てしまったわけで，**X68000** の単なる一ユーザにすぎなかった私にとってはまったく驚くべきことです。私がアセンブラに手をつけ始めてから 2 年余り，雑誌メディアやパソコン通信，その他いろいろな所で **X68000** ユーザのパワーを肌で感じることができましたが，考えてみればこのような企画が実現したのもそうしたパワーのたまものといえるわけで，その意味では改めて皆さんに感謝したいところです。

周囲を見れば 386 や 486 といった，一昔前までは考えもしなかったような高速のマシンが次々と発表されている中，いまだにメイン CPU が 68000 などというマシンが活躍しているのはほとんど奇跡といえるのかもしれません。しかし，それは奇跡でも何でもなくて，現実に今でも多くのユーザが思い思いの方法を使って **X68000** の世界を楽しんでいるわけです。そうした楽しみの 1 つに，本書が少しでも力になれば実にうれしいことだと思います。

中村祐一 (**HAS** 担当)

今回が初めての執筆なのですが，書き終わってみると自分ではかなりの量を書いたつもりが意外と書けていないものですね。この部分は，どうしようかなどと悩んでいるうちに時間だけが過ぎていき，いつのまにか締切りを過ぎてしまい，担当の方にはずいぶん迷惑をかけてしまいました。(^_^;

もともとドキュメント等を書くのが苦手だったので，自分の意図していたことが正しく伝わっているかどうか，若干の不安があったりしますが，どうでしょうか？ ユーザの皆さんが，本書によってリンカのトラブルで悩まれることが少なくなれば，筆者としてはうれしいかぎりです。

最後に，手紙やドネーション等を送ってくださった方々，ボードやメイルで感想等を書いてくださった方々，有難うございました。この場を借りてお礼を申し上げます。

石丸敏弘 (**HLK** 担当)

書き終わってみると，内容のほとんどがコマンドの説明になってしまいました。それは，デバッガというプログラム開発ツール自体がプログラムのデバッグを手助けするものであり，使い方といってもバグの種類や状態によって違ってくるものだからです。本書では，**GDB** のコマンドをデバッグ作業の用途ごとにまとめ，個々のコマンドをわかりやすく紹介したつもりです。ユーザの皆さんが自分のデバッグスタイルにあわせて，**GDB** の豊富なコマンドをうまく使いこなすことを願っています。

今野幸義 (**GDB** 担当)

う～，やっと脱稿できました。C マガジンやパソコン通信などで **GCC** は配布してきたわけですが，きちんとしたマニュアルがないコンパイラだとやっぱり使うのに不便ですね。それで，ようやく今回ちゃんとしたマニュアルが作成できました。

GCC はオリジナルから相当に拡張されているのですが，**X68000 GCC** では，かなり長い期間，パソコン通信で洗礼を受けたためにオリジナルにはない機能がいっぱいついてしまいました。当然，この部分はオリジナルマニュアルにはないので，初めて使うユーザには使い方がさっぱりわからない，しかも機能が追加になるたびに，簡単な説明をつけていただけなので過去の使い方を知っていないと全然使えないといった火に油を注ぐような，不親切なコンパイラへの道をひた走っていました。

このような状況をある程度打開するために，最初は Oh*!* X での “Z-MUSIC SYSTEM” のようなムック形式で **GCC** を配布できないかと打診したのが，本書の始まりです。企画の段階では **GCC** だけだったのですが，書籍にすることが決まった時点で **GCC** だけでなく，すでにパソコン通信の世界では標準ツールとなっている **HAS** と **HLK** を加え，さらに C の高性能なソースコードデバッガ **GDB** をも巻き込んで，開発環境としてお届けすることになりました。ちょうど同じ時期に，**X68000** の C 言語のライブラリの本を作成する企画も進行していたために，この 2 つを統合して一連のシリーズ物として発行する計画になったのです。計画はこのようになっていますが，実際に決まっているのは本書と次のライブラリだけです (わははは)。そうそう，**X68000** の次機種のうわさもあります。いちおうこのシリーズでも，この次機種への対応も考えていますので，そのときはまたよろしくお願いします。

GCCの部分と概説を私が担当しました。普段使っているコンパイラですから，使い方や拡張仕様の説明などの部分では苦にならなかったのですが，たぶんコンパイラマニュアルでは“初めて”と思われるエラーメッセージの解説部分が大変でした。誤訳が見つかるたびに，コンパイラのソースを修正して再度コンパイル。また，エラーを出力させるのも一苦労です。コンパイラのソースから該当エラーを出力するであろうエラー例を推測して作成，コンパイルして確認の繰り返しです。この作業中に，変なエラーソースをコンパイルするとコンパイラがバスエラーで昇天することがあるのがわかり，それを修正したりで，大変な手間がかかってしまいました。多分誤訳はもうないとは思いますが…。

もうひとつ，なぜ**GCC**がVersion 2ではないのかということですが，このバージョンは猛烈な勢いでバージョンアップが続いていることと，まだ安定性に欠ける面があるためです。

最後に，本書の作成に使わせていただいたフリーソフトウェアとその作者の方々に著者を代表して感謝いたします。

- `CONDRV.SYS`
  卑弥呼☆氏
- `TwentyOne`, `preview.x`, `print.x`, `fontman.x`
  Ｅｘｔこと川本琢二氏
- `Nemacs`
  村上敬一郎氏
- `GNU make.x`, `readlinelib.a`, `ptermcaplib.a`
  homy氏
- `readlinelib.a`, `ptermcaplib.a`, `h68unix.a`
  Mad Player氏
- `gnulib.a`
  大槻氏

ほかにもたくさんたくさん感謝の言葉もありません...

吉野智興(**GCC**担当)

X68k Programming Series #1

X680x0 Develop. Manual Books

1994年9月5日　初版発行

著　者　吉野（よしの）　智興（ちくみ）　　中村（なかむら）　祐一（ゆういち）
　　　　石丸（いしまる）　敏弘（としひろ）　　今野（こんの）　幸義（ゆきよし）

発行者　橋本　五郎

発行所　ソフトバンク株式会社出版事業部
　　　　〒103　東京都中央区日本橋浜町 3-42-3

　　　　TEL　販売　03(5642)8101
　　　　　　　編集　03(5642)8143

印　刷　東京書籍印刷株式会社

ISBN4-89052-533-5

Cover Design = Tetzya Yonetani
Style Design = Tateaki Hori

SOFT BANK ソフトバンク

ISBN4-89052-533-5
C0055 P5300E
9784890525331
2冊セット定価5,300円
（セット本体5,146円　分売不可）
1910055053007

X68k
Programming Series
(#1)
X680x0
Develop.